Amy Compiler Extension - Imperative Features

Compiler Construction '18 Final Report

Arthur Passuello - François Quellec

EPFL

arthur.passuello@epfl.ch - francois.quellec@epfl.ch

1. Introduction

Amy is a simple purely functional language that could be described as a more basic version of *Scala*. In its current version, there can be only final `val` variables but new classes and interfaces can be defined. The only *control-flow* operations are `if-then-else` and `match-case` conditional statements, along with *functions* definitions.

Thus, this language allows only for plain functional program structure, which this project aims to alleviate in some way by adding two *imperative* features : *mutable* variables `var` and `while` conditional statements.

Up until this last phase, our work along the semester was to implement each step carried out by the `Amy-compiler` in order to transform a source file, into a runnable *WebAssembly* byte-code (or a list of error), i.e *lexical analysis/parsing, type and name analysis* and *translation*.

The lexical analysis starts by *parsing* the input file into a list of recognized `Tokens`. This list is then scanned to construct the corresponding *Abstract Syntax Tree* using the grammar rules defined in the `Parser`, and the `AST` constructors. Note that the grammar had to be modified in order to make it *LL(1)*, which guarantees linear parsing time.

For the next step, name and type analysis, we had to transform the previously defined tree into a `Symbolic Tree` by associating every variable, function and class name to a unique `ID` - note that this step is also responsible for checking for naming consistency. We then had to implement the `TypeChecker`, which would read through this tree and generate a list of typing (e.g. function return type and variable type) `Constraints` [expected, actual]. This list is then skimmed and any inconsistency generates an error.

Finally, the `Symbolic Tree` previously computed is read through to generate a sequence of `WebAssembly` instructions. The use of the tables for the various stored aliases is translated into exchange with the memory stack. The stack then contains references to variables, function and class constructors definitions.

2. Examples

The following example describe two implementations of the same `Factorial` function in *Amy*. Although both examples require similar amount of code, it will be explained how both lead to different results and how their respective process flow is impacted.

```
/* Implementation using previously defined features */
object Factorial {
  def fact(i: Int) : Int = {
    if(i < 2) { 1 }
    else {
        i * fact(i − 1)
    }
  }
}
```

This rather simple function compute the factorial value of a given `Integer`. This function presents a recursive structure and will call itself as many times as required before the last call returns the value `1` and make all the other calls return like a waterfall. Each iteration, except the last one, is composed of 1 comparison, 1 subtraction, 1 multiplication and 1 function call.

```
/* Implementation using imperative features */
object Factorial {
  def fact(i: Int) : Int = {
    var fact: Int = 1; // Declaration
    var index: Int = 2;
    while(index <= i){
        fact = fact*index; // Reassignment
        index = index + 1
    }
    fact
  }
}
```

In this implementation, the `var` keyword is used to define two mutable `Integer` variables with initial values `1` and `2` respectively. Then the variable *index* is compared to the function's argument to express the current state of the function's processing. If `true`, the processing requires at least another step : the `fact` variable is updated and the `index` is incremented, the condition (`index <= i`) is then evaluated again. When `false`, the function goes to the next line after the `while` instruction's body and return the current value of `fact`. This implementation, as opposed to the first one, requires a single call to the function *fact* but requires the same number of steps, each including 1 comparison, 1 addition and 1 multiplication.

The mutable variables are declared in a similar way the immutable ones are, using only the keyword `var` instead. On the other hand, in the reassignment of those variables, a whole new usage of the "=" (`EQUALS` token) delimiter is made, it is used as an operator.

The condition provided after the `while` instruction is evaluated first, if it is *true*, then the associated block is executed and an unconditional branch leads back to the condition for another evaluation, otherwise, the program jumps directly to the line following the aforementioned block.

From those examples, both aspects of those implementations can be directly compared : in terms of algorithmic complexity, both require more or less the same amount of similar operations but in a more concrete perspective on the compiled result, we see that the second implementation, using imperative features, requires only 1 call to *fact* and only 2 variable assignments regardless of the argument's value. However, the first example, purely functional, requires a linear amount of call to *fact* with respect to the value of `i`.

3. Implementation

This section initially introduces the theoretical concept involved in the conception of the imperative feature. Then it describes more or less thoroughly each step of their implementation.

3.1 Theoretical Background[Kunčak 2018]

As mentioned in the first section, the process of compiling an *Amy* source file into a `WebAssembly` byte-code requires a thorough analysis of the program syntax and logic. This analysis relies on a formal description of the language, composed of `Tokens` and `Rules` declaration to parse and try to represent the program as a `tree` structure, both helpful in representing the static architectural aspect of the program and its process flow.

From this theoretical point of view, only the addition of the mutable variable `var` requires non-trivial modifications to the current version of the compiler. The `while` instruction would of course need its own new token, new rule and constructor but can then be described and analyzed the same way an `Ite` instruction would be.

The introduction of mutable variables on the other hand brings up the issue of variables scope and type. Variables initially implies a single `Id-Type-Value` description, to then only be reference by the `Id` parameter. Upon such a declaration, those 3 values would be stored in a table that would be used for different things :

- check that subsequent declarations do not use the same `Id` value (name)
- check that following reference to `Id` would be in the right context (type)
- provide the `Value` for computing purposes

Note : depending on the context of such variable declaration (main, functions, match-cases, ..) global and local tables are used, much like the `symbolic tree` representing the program.

This implementation, as is, does not allow for any modification of an existing entry in those tables, hence, in order to implement mutable variables, one could do either of the following :

- define and construct separate tables whose operations would allow modification of existing entries while providing the information necessary to ensure the correctness of such an operation $\rightarrow$ same structure than existing tables
- add an underlying parameter to the entries in the existing table that would indicate whether this variable should be modified or not, and therefore modify the tables usages everywhere.

Either way, in order to implement the `var` feature, the `Name Analyzer` must be provided ways of knowing whether a variable is mutable, along with its type. The `Type Checker` remains unchanged as the type cannot be modified. Finally, the `Code Generator` and its underlying data structure must provide a way to override the values of certain variables in their tables.

3.2 Implementation Details

To follow the development steps, this section will start by introducing the `variable` implementation to then detail the implementation of the `while` instruction.

3.2.1 Variables - VAR

As detailed in **Section 3.1**, this new feature can be defined using wide inspiration from the already defined `val` keyword and its usage throughout the compiler's different steps. This practical observation presents although its limitation and some modification were necessary to maintain the grammar's properties whilst correctly inserting these new elements. The declarations of `var` and `val` partly shares indeed the same structure but differ in the fact that a `var` might not be initialized at its declaration, as it can be later. Finally, it was necessary to introduce in the grammar the possibility of re-assigning values to `var` entities. The modified parts of the existing grammar for this matter are as follow :

```
'Expr ::= 'ExprMatch ~ 'ExprH
              | 'ExprVar,

'ExprVar ::= VAL() ~ 'Param ~ 'Assign ~ 'Sequence
              | VAR() ~ 'Param ~ 'ExprVarH ~ 'Sequence
              | 'VarId ~ 'Assign ~ 'ExprH,

'ExprVarH ::= epsilon()
              | 'Assign,

'Assign ::= EQSIGN() ~ 'ExprMatch,

'ExprH ::= epsilon()
              | 'Sequence,
'VarId ::= IDVARSENT
'Sequence ::= SEMICOLON() ~ 'Expr,
```

Using this grammar, the *Parser* can correctly interpret both declaration types of `var` and its reassignment but, in order to avoid parsing ambiguity - which will be further discussed in the next paragraph - it was deemed necessary to define new *Tokens*. This necessity comes from the fact that the *Analyzer* will need to be able to differentiate *Id* usage among functions, `val` and `var` expressions. Those new *Tokens* are defined as follow :

```
case class VAR() extends Token
case class ASSIGN(value: String) extends Token
                                        with TerminalClass
val IDVARSENT = ASSIGN("")
```

The first *Token* is rather obvious and comes from the necessity to define a token for the `var` keyword. The other two, less obvious, come from the fact that, as stated previously, an *Id* that corresponds to a `var` needs to be able to be (re)assigned. Yet, so far, expressions using *Id*s could correspond to `val` and function declarations or references. The need for a new usage for *Id*s (assignment) hence implied the need of a new *Token* to represent those *Id*s, that would be different from the existing one.

Without this early differentiation in the *Id* definitions, one key property of the grammar would be violated as *'Expr* would have two derivation trees using an *'Id* (function/variable call and variable assignation), i.e.

$$First('Expr) \cap First('Expr) = ID() \neq \emptyset$$

and would result in it not being *LL(1)* anymore. Hence, separating from the start *Id*s that can be assigned and *Id*s that can only be referenced following their definition removes the ambiguity and preserves our grammar's properties.

Then, in the *Lexer*, it was necessary to look ahead for the following characters of the expression, after recognizing an *Id* to determine whether it is one followed by an assignment (*ASSIGN* token), or a simple reference (*ID* token). In that purpose, it was decided to check the following:

$$nextChar \neq " = " \quad (1)$$

$$secondNextChar == " = " \quad (2)$$

$$secondNextChar == " > " \quad (3)$$

$$secondNextChar == "\{" \quad (4)$$

If one of the 1-4 equations is true, then it is an *ID* token, otherwise it's an *ASSIGN*.

Using those new elements, the compiler is now able to always correctly recognize references to variables that are destined to an assignment, and those that are mere references, thus closing the *parsing* chapter of the modifications for this feature. The compiler now needs a way to ensure that re-assignment to `val` should not be allowed, and should be for `var`.

The answer to this issue first resides in the *AST* construction. In order for the *Analyzer* to be able to determine whether an assignment is correct or not, a new field has been added to the *Identifier* object, *ASSIGNABLE* - false by default- and an overridden version of the *fresh* method, as follow:

```
val ASSIGNABLE = true

def fresh(name: String, assignable: Boolean): Identifier
                    = new Identifier(name, assignable)
def fresh(name: String): Identifier
                    = new Identifier(name, !ASSIGNABLE)
```

This structured is used in order to minimize to modifications to the existing compiler : previously defined calls to *fresh* will remain unchanged and have the effect of defining a non-assignable (i.e. immutable) *Id*. On the other hand, it is now possible to specify whether a new *Id* should be. Thus, the *NameAnalyzer* now specifies, upon translating an expression from a nominal program into a symbolic one, specifies that an expression of type `N.Var` initiate the creation of a symbolic *Id* using the *fresh* method as follow :

```
val idName = Identifier.fresh(df.name,
                              Identifier.ASSIGNABLE)
```

Upon translating an expression of type *N.Assign*, it ensures that (beyond verifying that the *Id* is defined) the *Id* is indeed assignable as follow :

```
val nameS = locals.get(name)
              .getOrElse(params.get(name)
              .getOrElse(fatal(s"Undefined variable name")))
if(nameS.assignable != Identifier.ASSIGNABLE)
    fatal(s"Variable name is not assignable")
```

The *Analyzer* is now able to determine the legality of a variable assignment, it only further requires to ensure that a `var` is initialized before it is referenced (`read-before-write` error). Considering a proper check of such situation would require the implementation of a new and unknown part of the compiler, the *Data-Flow Analyzer*, it has been decided that all `var` variable should have a default value, if not set at their declaration (e.g. 0 for `Int`, *false* for `Boolean` and *""* for `String`). This "short-cut" solution remains correct and is very efficient in ensuring that no uninitialized variable can be read.

Finally, once all those issues had found a resolution, the *Code Generation* for all the parts of the `var` usage went rather smoothly and the result was what was anticipated.

3.2.2 While Instruction

Compared to the first part of this feature extension, the implementation of the `while` instruction was quite straightforward. It was decided that its global behavior should follow the `if-then-else` structure, i.e. using a `boolean` for *condition* but more importantly, having a *body* of `Unit` type. This second choice, comes from the fact such type of instructions have for sole purpose to modify the *process-flow* of the program and no correct or intuitive usage would require its body to return anything.

Hence, the implementation, although it required only to add the necessary parts to the *Lexer*, the *Parser*, the *ASTConstructor*, the *NameAnalyzer*, the *TypeChecker* and the *CodeGenerator*, was quite intuitive and followed the anticipated path.

One notable part of those additions though was the code generation : if there was analogous instructions in *WebAssembly* for the `if-then-else` instruction, it was not the case for the `while`. Fortunately, the latter can be constructed using the first - and a branch instructions. Using the idea that a `while` instruction is nothing else than an `if-then-else` whose `then` body loops back to the condition and `else` body simply is the continuation of the program. The translation of this instruction and its impact on the *WebAssembly* resulting code can be described as follow

```
val loop_start = getFreshLabel()
val loop_end = getFreshLabel()
Block(loop_end) <:> // BLOCK WHILE
    Loop(loop_start) <:> // LOOP BODY OF WHILE
        cgExpr(cond) <:> If_void <:> // if condition
            cgExpr(body) <:> // execute body
            Br(loop_start) <:> // jump to loop_start
        Else <:>
            Br(loop_end) <:> // jump end of loop_end
        End <:>
    End <:> // END LOOP BODY OF WHILE
End <:> Const(1) // END BLOCK WHILE
```

3.2.3 Tests implementation

In order to ensure that the added features did not break the existing version of the compiler, and respected the provided specifications, it was decided to implement a series of tests. Those were added to the existing `/test` folder in the directories that were deemed pertinent and, although some were there to merely check that the parsing was done correctly, some were implemented to ensure that the features had the expected behavior and raised the correct errors when needed. Although it was possible to check automatically the latter, the resulting behavior of a correct use of the new features was done by compiling the two files presented below, running them both using `nodejs` and verify that the printed values on the terminal corresponded to what was expected.

Note that the syntax correctness checks (i.e. that one cannot assign new values to `val`, assign values of a different type to `var`, the return type of `while` is *Unit()* and the default value of each primitive type is correct) were done by writing erroneous expressions in test files that were specified as expected to fail in the Scala test suite.

```
object Variable {
    var i: Int;
    var j: Int = 0;
    i = j;
    j = i + 1; }
```

```
object WhileLoop {
    def fact(n: Int): Int = {
        var res: Int = 1;
        var j: Int = n;
        while(1 < j) {
            res = res * j;
            j = j − 1
        };
        res
    }
}
```

4. Possible Extensions

Those features could be extended by other *flow-control* instructions such as `for-each` loops without having much to change to the version updated in this report, although the advantages of such features would be debatable. The same could be said for implementing unary operators "++" and "- -" for mutable variables.

What would on the other hand make the language much more convenient would be to implement type inference on mutable variables `var` along with mutable types, to allow expression of the form

```
var i = 1;
i = 'a';
i = Map(1 −> 'c')
```

But would reject the following :

```
var i = 0;
i = 'abc';
i = i + 1; // Type Error
i = i ++ 'defg'; // Ok
```

This feature would raise questions about the scope of such modifications and would require to further modify all the work done in this last phase to allow more modification in the variable tables. This seems, however, like a great addition to the language.

References

V. Kunčak. Computer Language Processing. EPFL, 2018.