Assignment 3
Translation Report

COMP3000 Programming Languages 2020

Written by Joel Morrissey (45238049)

Contents:

Introduction:

Overview/Aim:

This assignment had us create a syntax translator in Scala for a new language called FunLang. This translator will take a source tree as input and compile it to the given SEC machine's instructions. This language consists of expressions, blocks, conditionals expressions/statements, binary operations and function definitions. Through this task new knowledge and appreciation of translators has grown.

Structure of this report:

This report shall cover all the project details including the design, implementation and testing of the application. During the discussion of the design and implementation of the project, we will look into why and what choices were made to translate the given program (source tree) and how these designs were implemented into code. During the testing portion of this report, we will detail into the steps taken to test the program and discuss why these tests should be sufficient to prove the validity of the program.

Design and implementation:

Translating constants and use of identifiers

```
73              case IntExp (value) =>
74                  gen (IInt (value))
75
76              case BoolExp (value) =>
77                  gen (IBool (value))
78
79              case IdnUse (value) =>
80                  gen (IVar (value))
```

Design:

These are translated very simply as they simply required instructions to be put on top of the stack.

Implementation:

To do this the values were converted directly to SEC machine instructions with the use of gen to append it to the instruction buffer.

Translating binary operations and conditional expression:

```
82              case PlusExp (l, r) =>
83                  genall (translateExpression (l))
84                  genall (translateExpression (r))
85                  gen (IAdd ())
86
87              case MinusExp (l, r) =>
88                  genall (translateExpression (l))
89                  genall (translateExpression (r))
90                  gen (ISub ())
91
92              case StarExp (l, r) =>
93                  genall (translateExpression (l))
94                  genall (translateExpression (r))
95                  gen (IMul ())
```

```
97              case SlashExp (l, r) =>
98                  genall (translateExpression (l))
99                  genall (translateExpression (r))
100                 gen (IDiv ())
101
102             case EqualExp (l, r) =>
103                 genall (translateExpression (l))
104                 genall (translateExpression (r))
105                 gen (IEqual ())
106
107             case LessExp (l, r) =>
108                 genall (translateExpression (l))
109                 genall (translateExpression (r))
110                 gen (ILess ())
```

Design:

With each of these matches we needed to put the left side of the operand on the stack first followed by the right side. Then after these two values were put onto the stack then the relevant instruction needed to be put onto the

stack so that both the values of the right side then the left side would be popped of the stack and the resulting value from the operand would be placed back on the stack.

Implementation:

The implementation of this was also very simple with each translating the right side of the operand first then the left side. These all use the genall function along with the translateExpression function to translate expressions within expressions. An example of this is (5 * 2 + 1 - 3) notice with this example it will need to translate on either side of the plus being to translate 5*2 (for the left) and 1-3 for the right before it translates the value of the plus. Once both the left and right side of the operand has been translated their values will now be on the stack. Then the program will but relevant operand instruction onto the stack using the gen function which will pop the results of the two sides of the operand off the stack and apply the operation given.

Translating conditional statements (If statements)

```
112             case IfExp(c, t, e) =>
113                 genall (translateExpression (c))
114                 gen(IBranch(translateExpression (t), translateExpression (e)))
115
```

Design:

During the designing of how to handle if statements we know that we need push the value of the condition onto the stack first then put an IBranch containing the frames of the true and false sections of the if statement. The SEC machine will make the relevant jumps to the given frame depending on weather the top value is true or false.

Implementation:

The implementation starts with translating the condition, it uses genall and translateExpression to ensure that the expression (c) is translated fully and then we know that the resulting value added to the stack (this will either be true or false). Then an IBranch instruction placed onto the stack with the true frame (t) and the false frame (e). This is all we needed to do because as mentioned above the SEC machine will pop of the IBranch and a condition and jump to the relevant frame of IBranch depending on the condition given.

Translating use of functions

```
116             case AppExp(f, arg) =>
117                 genall (translateExpression (f))
118                 genall (translateExpression (arg))
119                 gen(ICall())
```

Design:

During the design phase of translating application expressions (a call of a function) we know that the ICall requires to pop off an argument value

followed by a function name. Meaning that we must first put onto the stack a function name then it's arguments.

Implementation:

For the implementation of this, genall and translateExpression is used to create the series of instructions required to push on the function name (f) followed by the instruction for the argument (arg). Once this is done it is expected that the value of the name is underneath the value of the argument in the stack. The program then puts onto the stack an ICall using the gen function to then pop of the argument value and then the function name from the stack. This will then call the given functions with the given argument value.

Translating a block

```
121             case BlockExp(head +: defns, body) =>
122                 //match all the different possible block heads
123                 head match {
124                     case Val(IdnDef(idn), exp) =>
125                     {
126                         genMkClosure(idn, BlockExp(defns, body))
127                         genall ( translateExpression(exp) )
128                         gen (ICall())
129                     }
130                     case FunGroup(Fun(IdnDef(idn), Lam(Arg(IdnDef(x),_), exp)) +: fundefns) => {
131                         genMkClosure(idn, BlockExp(FunGroup(fundefns) +: defns, body))
132                         genMkClosure(x, exp)
133                         gen (ICall())
134                     }
135                     case FunGroup(Vector()) => {
136                         genall ( translateExpression(BlockExp(defns, body)) )
137                     }
138                 }
139 
140             case BlockExp(Vector(), exp) => {
141                 genall ( translateExpression(exp) )
142             }
```

Design:

When designing how to translate a block expression, we needed to understand how to handle a block. As described in the assignment specification, we handled blocks as if they were function calls. To do this however we need to consider all the difference cases that could occur being that the head element could be a Val or a FunFroup. We also knew that if values keep getting removed from the vectos of either the BlockExp or the FunGroup then the program would need a way to catch those cases and handle them appropriately. It should also be noted that genMkClorsure was used to create the IClosures as it handled the creation and removal of different environments.

Implementation:

The final two cases match Block expressions with all the cases above in mind.

The first case on line 121 match any block expression with a vector that is not empty. When it matches this block expression it maps the head and tail (defns) of the vector into two separate variables for use later. A match on the

head is then ran to discover weather the first definition of the block is a Val or a function group (FunGroup).

If the head is a Val, then the identifier (idn) and what that identifiers value is being set to (exp) is extracted. An IClosure is then made using the genMkClorsure passing in the identifier and a new BlockExp containing the rest of the vector (defns) (the originals block expressions vector without the head value) along with the final element of the block (body). Once the IClosure is created the expression (exp) is translated using genall and translateExpression (the resulting value pushed onto the top of the stack). Finally, an ICall is placed onto the stack to call the IClosure with the value of exp to get the result of that block we created. It should be noted that the code on line 126 where a new BlockExp is created without the head element in the vector allows for many definitions to be translated in the block as genMkClorsure will try to generate the code for the rest of the BlockExp.

If the value is a FunGroup then it will extract the head function of the FunGroup and similar to when Val extracts out the name (idn) of the function and the expression it maps to, the match to FunGroup also extracts out the name of the argument (x) the function accepts. The rest of the functions are held in a variable named fundefns. Also similar to Val we call the genMkClorsure function with the function name (idn) and a new block expression, however this time the block expression contains a FunGroup with the rest of the functions of the FunGroup that was just matched combined with the rest of the definitions (defns) and the body expression. genMkClorsure is then called again to create another IClosure which maps the name of the argument (x) to the body of the function (exp). Finally, an ICall is placed on the stack which will pop of the IClosure created on line 132 and then call the IClosure created on line 131 with the IClosure on line 132. The case on line 135 will catch when the FunGroup is empty and remove it from the blockExp and then call genall with translateExpression to translate the new block without the empty FunGroup inside it.

Finally, the case on line 140 will catch the case when there are no more definitions in the block expression, then the program will simply translate the final expression in the FunGroup. After which the entire block will have been translated and the block will be executed like a function and the resulting value will be placed onto the stack.

Testing:

Arithmetic and constant number expressions: (lines 63 – 131)

These tests test to ensure that all arithmetic and constant number expressions correctly get translated. These arithmetic expressions include “+”, “-“, “*” and “/”.

The first two tests check to ensure that if the program consist of a single constant number then it translates and evaluates to the correct result.

The next four tests check to see that all the binary operations (arithmetic operations) translate correctly with simply one integer on each side of the arithmetic expression.

The final four tests check to see that when these operations are chained together, order of operation (this is handled by syntax Analysis however still good to test here) is maintain and that they still correctly evaluate to the correct result.

From all this it is clear to see that if all these tests pass then arithmetic operations (tests for with variables is tested later in the block section) and constant numbers will evaluate correctly.

Conditional expression and conditional constants (lines 134 – 188)

These tests check that the program correctly translates conditional expressions, these include “<”, “==”, “true”, “false”.

The first two tests ensure that condition constant work correctly. These tests are quite simple in nature as there is only two cases being that of the conditional constant “true” and the conditional constant “false”.

The next four tests are used to ensure that simple conditional expressions (one term on each side) work correctly. These tests mainly ensure that “<” and “==” correctly (and can) evaluate to true and false.

The final two tests ensure that complex expressions still work on either side of the expression and still evaluate to the correct Boolean constant after evaluation.

From these tests it is clear to see that if these tests are passed then conditional expressions are translated correctly.

Conditional statements (if statements) (lines 191 – 245)

These tests ensure that conditional expressions/constants work correctly with if statements and that the correct expression will be evaluated depending on whether or not the conditional statement is true of false.

The first two tests are to ensure that the both the “true” and “false” constants in an if statement evaluate correctly and that the correct branch is taken depending on which is given.

The next four tests ensure that simple conditional expressions (only one term on each side of the conditional expression) are evaluated correctly and cause the correct branch to be taken.

The final two tests ensure that complex terms work in the results from the if statement as well as in the conditional statement. It should be notes later tests will cover the inclusion of blocks and more complex conditional statements.

From this it is clear that if these test pass then conditional statements are correctly translated and executed by the SEC machine.

Block expressions (lines 248 – 337)

These tests ensure that blocks are translated correctly as well as some more complex versions of previously tested language features work correctly.

The first two tests are simple and just ensure that block expressions are correctly evaluated with both simple functions and simple variables.

The next test contains a variable and function definitions in random order to ensure that functions and variables still will be translated and result in the correct output.

The next test begins to test more complex expressions in blocks such as functions and variables which rely on conditional statements. It also tests that one function can call an earlier created function. Lastly this test checks to make sure that blocks can perform arithmetic operations with other blocks and expressions.

The final test tests many things including complex conditional statement, nested blocks and accessing parent environments. It tests to ensure that all expressions can have blocks within them such as a block in an if statement, a function and a variable. It also checks to ensure that if statements can be called within each other. It also ensures that functions can be called to provide Boolean values with one example taking a Boolean in and returning it and another example taking an integer and returning weather it is ten of not. It also tests to see if a function can be called inside an argument of another

function. Lastly it tests to ensure that variables in parent scopes can be accessed in children scopes.

From the outline of these tests it is clear how all the test cases are covered and ensures the program is translated correctly for block expression and complex versions of earlier features tested.