DEPARTMENT OF COMPUTER SCIENCE, TECHNISCHE UNIVERSITÄT KAISERSLAUTERN
COMPILERS AND LANGUAGE PROCESSING TOOLS - SS17

A Minijava to μ-LLVM Translator

Exercise 4
Group 03
Daniele Gadler, Gopal Praveen
Alka Scaria, Stephen Banin Payin

June 14, 2017

OVERVIEW

In the present report, we describe the Translator from Minijava to μ-LLVM. The resulting Translator is able to translate the following statements and expressions: Block, StmtIf, StmtWhile, StmtReturn, StmtPrint, StmtExpr, StmtAssign, ExprBinary, ExprUnary, BoolConst, VarUse, Number, as well as ArrayLookup, ArrayLength, ExprNull, NewIntArray.
The translator passes both all test cases provided in the exercise' template and all test cases laid out by the group mates covering further edge cases.
We implemented our solution through the visitor pattern, breaking down the implementation into different classes for order, understandability and especially functionality purposes. Following, we describe the classes' role in the program.

CLASSES' DESCRIPTION

- **Translator**: Contains the Default Visitor Pattern for the following principal Minijava expressions and statements: MJStmtIf, MJBlock, MJStmtWhile, MJStmtPrint, MJVarDecl, MJStmtAssign. Content of these expressions and statements is handled separately in other classes, again through the visitor pattern.
  This class also contains the current block into that expressions are to added and the Hashmap for variables' declarations.
- **ExprMatcherR**:Implements the visitor pattern for "MJExpr" . It matches expressions that can occur on the right-hand side of an assignment operation (e.g: MJExprBinary, MJExprNull in an array assignment, MJExprUnary for MJUnaryMinus and MJNegate, MJBoolConst, MJNumber, MJVarUse for different types, MJNewIntArray, MJArrayLength, MJArrayLookup).

- **ExprMatcherL**: Implements the visitor pattern for "MJExpr". It matches expressions that can occur on the left-hand side of an assignment operation (e.g: MJVarUse with different types and MJArrayLookup).
- **OperatorMatcher**: Implements the visitor pattern for "Operator" and the operation type being used in a binary expressions (e.g: MJPlus, MJMinus, MJLess, MJAnd, MJDiv, MJTimes).
- **StaticMethods**: Contains static methods being invoked repeatedly throughout the program for handling operations on arrays.
- **TestOutputLLVM**: Class used for testing the program and invoked by the "Main" class in the "frontend" folder. It compares the output of the LLVM-generated program with the Minijava program and checks whether these are equal.

TECHNICAL DESCRIPTION

TASK 1 - TRANSLATION OF STATEMENTS AND EXPRESSIONS

- **Block**: A Block is a handled as an iterable list of statements. Each statement is matched to its type in the Translator class.
- **StmtIf**: An If-else statement is handled through three different Basic Blocks, corresponding respectively the "IF body" (condition true), the "ELSE body" (condition false) and the remaining code. Initially, the if condition is checked and a Branch operation picks the Basicblock where execution should continue on. After evaluating expressions and statements containing in either of these blocks, execution continues in the block for the remaining code.
- **StmtWhile**: A While statement is also handled through three different Basic Blocks, containing respectively the condition to be checked, the while loop's body and the rest of the code. In every iteration, the condition block containing a Branch statement with the while's condition is checked: if the condition is true, the statements contained in the loop's body are evaluated (afterwards, we jump back to the condition block and check the condition again). If the condition is false, we simply continue with the basic block for the remaining code outside the while.
- **StmtAssign** We handle assignment statements by matching expressions that can occur on the left-hand side (ExprMatcherL) and expressions that can occur on the right-hand side (ExprMatcherR) by making use of the visitor pattern and overriding the adequate cases as outlined in the Classes Description. Expressions being matched on the left or right hand side that are not allowed in the current implementation (e.g: MethodCall or ExprThis throw an error). We also make use of a class "OperatorMatcher", which returns the operator being used in the expression being translated. We then store the value of the expression matched on the right-hand side into the left-hand side.
- **StmtPrint**: In a print statement, we simply check for it to contain expressions that can occur on the right-hand side of an assignment expressions and print it out.
- **Variable Declaration**: We allocate all newly declared variables onto the stack with space depending on their type also store them into a HashMap.
- **VarUse** Whenever making use of a variable, we distinguishing between its usage on the left or right hand side (left means store into it, right means load from it). In both cases, we still need to get the corresponding temporary variable definition from the Hashmap and the

type, which is obtained from the extra information provided by the previous phase (Type and Name Analysis).

- **UnaryMinus**: We implemented the unary minus expression (ex: -b) through a binary expression, where the expression with a minus in front of it is multiplied by -1.
- **UnaryNeg**: We implemented the unary negation expression (ex: !b) through a XOR with a true value so as to invert the boolean value.

Task 2 - Translation of Arrays

- **NewIntArray**: Array instantiation (ex: a = new int[5]) was implemented as suggested in the lecture slides of the course. Let s = array size (ex: 5), b = size of each element in (ex: ConstInt(4)). Since we will need to store the array size in position 0, we will then need to allocate the following amount of space (p) onto the heap:

$$p = (s+1) * b$$

More specifically, we allocate $s+1$ blocks of size b in contiguous heap space for the storage of elements and the array size. Let $e_1 ... e_n$, where $n < s$ be a list of elements that will be contained in the array at a later time: these elements will be stored in the array according to the following scheme:

value	s	e_1	e_n
position	0	1	n

In order to prevent LLVM from generating random integer values when performing an ArrayLookup operation on an instantiated array containing no value yet, we instantiate all blocks $1...n$ of an array to integer value 0. Finally, we check that the array is not being instantiated with a negative value: this case would rise an HaltWithError exception in LLVM.

- **Array Length**: Since the length of the array was stored at position 0, we simply get the array address get the the first value stored on the heap corresponding to the array.
- **Array Lookup**: Whenever looking up a value in an array, we check for two cases not to be matched:
  1. **Negative Index**: We ensure a negative index was not passed and throw an error in case that happens.
  2. **Out of bounds index**: We ensure we do not access a value that is beyond the array size by ensuring that for an index i the following always holds: $i < s$ and throwing and error should this condition not be met.

  Should these two cases not be met, then the elemented is retrieved from the array by increasing the index by 1 (since the length is contained at position 0).

References

[1] We would like to thank our study colleague Joseff for support with bitcasting arrays into the right format.