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 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 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 Default Visitor Pattern for the expressions and statements: MJStmtIf, MJBlock, MJStmtWhile, MJStmtPrint, MJVarDecl, MJStmtAssign. The class also has current block into which expressions are added and the Hashmap for variables' declarations.
- **ExprMatcherR**: Implements the visitor pattern for "MJExpr" . It matches expressions that occur on the right-hand side of an assignment operation (e.g: MJExprBinary, MJExprNull, MJExprUnary, MJArrayLookup).
- **ExprMatcherL**: Implements the visitor pattern for "MJExpr". It matches expressions that occur on the left-hand side of an assignment operation (e.g: MJVarUse and MJArray-Lookup).
- **OperatorMatcher**: Implements the visitor pattern for "Operator" and the operation type 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 arrays.

- **TestOutputLLVM**: Class used for testing the program and invoked by the "Main" class ("frontend"). 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 handled as an iterable list of statements. Each statement is matched to its type in the Translator class.
- **StmtIf**: There are three Basic Blocks, "IF body" (condition true), "ELSE body" (condition false) and remaining code. Initially, if condition is checked and a Branch operation picks the corresponding Basicblock for execution. After evaluation of the Basicblock, execution continues for the remaining code.
- **StmtWhile**: This also has three Basic Blocks, condition to be checked, while loop's body and rest of the code. The while's condition is checked: if true the condition block with a Branch statement to the while's condition is evaluated. If the condition is false, continue to remaining code.
- **StmtAssign** Handled by matching expressions that occur on left-hand side (ExprMatcherL) and right-hand side (ExprMatcherR) by using the visitor pattern. Invalid Expressions in the current implementation (e.g: MethodCall or ExprThis) throws an error. The class "OperatorMatcher", returns operator used in the expression. The value of matched expression on right-hand side is assigned into the left-hand side.
- **StmtPrint**: In print statement, we simply check if the expression to be printed is an Integer.
- **VarDecl**: We allocate all newly declared variables onto the stack with space depending on their type and also store them into a HashMap.
- **VarUse** We distinguish between the usage on left (store) or right (load) hand side. In both cases, get the corresponding temporary variable definition (Hashmap) and the type (Type and Name Analysis).
- **UnaryMinus**: Implemented the unary minus expression (ex: -b) through a binary expression, where it is multiplied by -1.
- **UnaryNeg**: Implemented the unary negation expression (ex: !b) through a XOR with a true value to invert the boolean value.

Task 2 - Translation of Arrays

- **NewIntArray**: Array instantiation (ex: a = new int[5]) was implemented as per the lecture slides. Let s = array size (ex: 5), b = size of each element (ex: ConstInt(4)). Since the array size is assigned to position 0, the amount of space (p) allocated for the heap is:

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

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

value	s	e_1	e_n
position	0	1	n

Inorder to prevent LLVM from generating random memory address on ArrayLookup for an instantiated array containing no value yet, we instantiate all blocks $1...n$ of an array to default integer value 0. If the array is instantiated with a negative value: the case would rise an HaltWithError exception in LLVM.

- **ArrayLength**: Since the length of array is stored at position 0, we simply get the array address and its first value from the heap storage.
- **ArrayLookup**: On looking up for a value in an array, we check for these two cases.
  1. **Negative Index**: We ensure that a negative index was not passed and if so throw an error.
  2. **Out of bounds index**: We also ensure that a value beyond the array size is not accessed by an index i that always holds: $i < s$ and throwing an error should this condition be false.

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

References

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