Transcompiler Documentation

CMPE-230, Systems Programming

TRANSCOMPILER

Kaan Yolcu 2020400150 & Enes Sait Besler 2020400159

01/05/2023

1 - Introduction

We implemented and converted the advCalc interpreter code, which we had previously written in the C programming language, into a transpiler that converts the advCalc++ language to the LLVM IR language.

Transcompiler: Transpiler that translates input in the form of assignment statements and expressions of the AdvCalc++ language into LLVM IR code that can compute and output those statements.

The commands required to run the program, the data structures of the program, the functions in it and the way they work are explained in an explanatory way in the continuation of the document.

2 - Program Interface

The user should compile the C code according to the Makefile we created for this. Makefile defines set of tasks to be executed.

The execution of the program will be done through the interpreter screen working in the terminal - in other words, it won't be a file-based program.

User will take an IR code, file.ll file as an input and should compile with lli command. That will give an output as file.s file. Then user will compile the file.s file with clang and will run the executable file.

```
>
./advcalc2ir file.adv
lli file.ll
llc file.ll -o file.s
clang file.s -o myexec
./myexec
```

3 - Program Execution

User will use an transcompiler for an advanced calculator.

We should summarize the IR code that we will use in the program as follows:

• LLVM IR uses static single assignment (SSA) based representation. In assignment statements, variables are assigned a value once.

- alloca is used to allocate space for variables and return address of allocation.
- Variables start with the % sign
- The i8, i16, and i32 keywords mean 8 bit, 16 bit, and 32 bit types, respectively.
- The * character denotes a pointer, just as in C.
- Integer variable names denote temporary variables.
- The IR contains the following piece of code, which defines the module name and the prototype for the printf function. You should use & generate this part as is in your translated IR files. ; ModuleID = 'advcalc2ir' declare i32 @printf(i8*, ...) @print.str = constant [4 x i8] c"%d\0A\00"
- To print the value of a variable you can can use the printf function we've defined above. call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %7)

The input .adv files will use a language that has the following properties:

- Every value and every calculation will be integer-valued (divisions should be rounded as it is done in C - i.e. 8 / 3 should be equal to 2).
- There will be no expressions or assignments with a result that exceeds a 32-bit number.
- Similarly, every bit-wise intermediate operation will abide by the 32-bit limit.
- The language does not support the unary minus (-) operator (i.e x = -5 or a = -b is not valid). However, as can be seen above, the subtraction operation is allowed.
- The variable names will consist of lowercase and uppercase Latin characters in the English alphabet [a-zA-Z].
- Expressions or assignments will consist of 256 characters at most.
- '%' characters denote comments. Any characters after '%' will be considered as a comment, not code.
- You must run and test your code on an Ubuntu Linux machine before submitting. You can use a Linux virtual machine or WSL in this context.
- The input advcalc++ language may include all sorts of syntax errors.
- Unlike advcalc, undefined variables cause an error in advcalc++.
- In case of syntax errors or undefined variables in the files, your output(s) should report them to the terminal in following form:

Error on line 8! Error on line 13!

- All of the following can be given as an input - and all of them are valid: a + b, a+ b, a + b, a+b, a +b, (((a))) + (b)

4- Input and Output

This is the inputs and outputs for Transcompiler. We tried to include as many different possible scenarios as we could.

Example Input 1 (From the project description):

```
a = 8
b = 8 * (a - 6)
c = b + 6
ls(a, 2)
xor(b - a + 1, 17)
y = 2 * b - c
xor(rs(y, 2), not(c))
```

Example Output 1 (From our program):

```
; ModuleID = 'advcalc2ir'
declare i32 @printf(i8*, ...)
@print.str = constant [4 x i8] c"%d\0A\00"
define i32 @main() {
%a = alloca i32
%1 = add i32 0, 8
store i32 8, i32* %a
%b = alloca i32
%2 = add i32 0, 8
%3 = load i32, i32* %a
%4 = add i32 0, 6
%5 = sub i32 %3, 6
%6 = mul i32 8, %5
store i32 %6, i32* %b
%c = alloca i32
%7 = load i32, i32* %b
%8 = add i32 0, 6
%9 = add i32 %7, 6
store i32 %9, i32* %c
%10 = load i32, i32* %a
%11 = add i32 0, 2
%12 = shl i32 %10, 2
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0)
%14 = load i32, i32* %b
%15 = load i32, i32* %a
%16 = sub i32 %14, %15
%17 = add i32 0, 1
%18 = add i32 %16, 1
%19 = add i32 0, 17
%20 = xor i32 %18, 17
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0)
```

```
%y = alloca i32
%22 = add i32 0, 2
%23 = load i32, i32* %b
%24 = mul i32 2, %23
%25 = load i32, i32* %c
%26 = sub i32 %24, %25
store i32 %26, i32* %y
%27 = load i32, i32* %y
%28 = add i32 0, 2
%29 = ashr i32 %27, 2
%30 = load i32, i32* %c
%31 = xor i32 %30, -1
%32 = xor i32 %29, %31
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0)
ret i32 0
}
```

Example Input 2 (That is our input for binary operations and call.)

```
x = 1
y = x + 3 % 4
z = x * y * y*y
z
lr(ls(rs(xor(((x)), x) | z + y, 1), (((1)))), 1)
```

Example Output 2 (From our program):

```
; ModuleID = 'advcalc2ir'
declare i32 @printf(i8*, ...)
@print.str = constant [4 x i8] c"%d\0A\00"
define i32 @main() {
%x = alloca i32
%1 = add i32 0, 1
store i32 1, i32* %x
%y = alloca i32
%2 = load i32, i32* %x
%3 = add i32 0, 3
%4 = add i32 0, 4
%5 = srem i32 3, 4
%6 = add i32 %2, %5
store i32 %6, i32* %y
%z = alloca i32
%7 = load i32, i32* %x
%8 = load i32, i32* %y
%9 = mul i32 %7, %8
%10 = load i32, i32* %y
%11 = mul i32 %9, %10
%12 = load i32, i32* %y
```

```
%13 = mul i32 %11, %12
store i32 %13, i32* %z
%14 = load i32, i32* %z
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0)
%16 = load i32, i32* %x
%17 = load i32, i32* %x
%18 = xor i32 %16, %17
%19 = load i32, i32* %z
%20 = load i32, i32* %y
%21 = add i32 %19, %20
%22 = or i32 %18, %21
%23 = add i32 0, 1
%24 = ashr i32 %22, 1
%25 = add i32 0, 1
%26 = shl i32 %24, 1
%27 = add i32 0, 1
%28 = shl i32 %26, 1
%29 = sub  i32 32 , 1
%30 = ashr i32 %26, %29
%31 = or i32 %28, %30
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0)
ret i32 0
}
```

5 - Program Structure

For our advanced calculator, we take inputs in string format and we assign this input to the lexer function we created in order to not tokenize these inputs according to the token types we have previously determined, and we keep them in the tokenlist structure we have created. Afterwards, our Expressioni parceling algorithm takes over and we create an Abstrac Syntax Tree (AST) by looping through our ParseBase functions recursively. There are functions and variables in the leaf nodes of this tree. Intermediate nodes have operation and equal tokens. Later, while parsing, expressions are calculated through recursion in the tree.

Since we used the same program structure that we used in the advCalc assignment, we changed the other parts of the document, but after 5.5, we started to add the structure part of the new assignment in the program structure section. ### 5.1 - Lexer

```
void lexer(char* input ,TokenList* tokenList)
```

We traverse the input we receive as a char array, and with the necessary if conditions, we create our tokens and add them to the tokenList with the void addToken(TokenList* list, Token* token) function we wrote before. The troublesome things here were space, integer and variables. We used the if statement

in the conditions where we saw space. We kept the number blocks together until a different character came into the integers. When we saw alphabetic characters, we first checked if they matched functions, if the function name was the tokentype, we recorded them in the tokenlist, and if it was a variable name, we created and saved it as a variable token. The remaining characters + , * .. were easy to check because they took up one byte.

5.2 - Parse Functions and Evaluation Process

```
Expression* parseBase(TokenList* tokenlist);

Expression* parseBase2(TokenList* tokenlist);

Expression* parseBase3(TokenList* tokenlist);

Expression* parseBase4(TokenList* tokenlist);

Expression* parseBase5(TokenList* tokenList);

Expression* parseExpression(TokenList* tokenlist);
```

First of all, we checked whether the expression specifies an equality expression in the parseExpression function. If there is equality, we looked to the right and left of the equality in a recursive way. We checked whether there is a variable on the left of it and determined whether there was an error condition. In the right part, we parse the expression in a recursive way by processing the same process as the case where parseExpression is not equality. In this process, we created 5 different parseBase functions due to processing priority.

```
Expression* parseBase5(TokenList* tokenlist) -> OR

Expression* parseBase4(TokenList* tokenlist) -> AND

Expression* parseBase3(TokenList* tokenlist) -> MINUS and PLUS

Expression* parseBase2(TokenList* tokenlist) -> MULTIPLY

Expression* parseBase(TokenList* tokenlist)  -> INSIDE PARANTHESIS and FUNCTIONS
```

We created the order of recursive functions according to the processing priority. Accordingly, we examined the situations according to the above-mentioned order.

Since the basic structure of Expressions consists of INTEGER and VARIABLE, we analyzed it in ParseBase and kept its values as parameters. In this process, we divided the function and at the same time kept the values of the expression as parameters, we performed the calculations.

We used the bool IsAlreadyVariable(char* var) function to see if the variable was previously assigned or not. If it is not assigned, we keep two arrays for this and assign values to the first space of the arrays. We called the long long int getValue(char* var) function for a previously assigned variable. If there is an assignment status, we have updated the value at that position of the array.

5.3 Data Types

These were the data structures we created to create the transactions we mentioned above.

```
typedef struct {
TokenType type;
char* string;
} Token;

typedef struct {
Token* tokens;
int numTokens;
int currentToken;
bool hasError;
} TokenList;

typedef struct {
TokenType type;
long long int value;
char* var;
struct Expression* left;
struct Expression* right;
} Expression;
```

5.4 Helpers and Error Logs

In order to determine the error logs, when we kept the hasError flag in the tokenList struct and detected an error, we logged it in the main function and ignored the operations done before, and continued the process of taking input again.

We detected foreign characters entered by the user with the bool isStringValid(char *string) function.

We wrote the necessary if statements for the program to stop and deleted the spaces at the beginning of the input after receiving the input. ### 5.5 LLVM Code Generation

5.5.1 - Memory allocation In the section where we first saw a variable in the previous advCalc code and created its expression, we generate the LLVM code here by using the “alloca” keyword.

```
fprintf(outputFile,"%%%s = alloca i32\n", tokenList->tokens[tokenList->currentToken].string
```

5.5.2 - Load and Store When we see the previously defined variable, we load its value from memory and assign a register value. We do this Variable expression type variablesa in ParseBase function. And for the left of equality in assignment expressions, we store the variable to the right of the equality regardless of whether it has been assigned before or not.

```
fprintf(outputFile,"%%%d = load i32, i32* %%%s\n", registerNumber, tokenList->tokens[tokenLi
fprintf(outputFile,"store i32 %d, i32* %%%s\n", next_expr->value, expression->var);
```

5.5.3 - Arithmetic Operations At first, if our expression is a number, we assign it a register value as "add" 0 value. Otherwise, we have trouble in cases where we print numbers. Then, in the next line of the createExpression parts of the parseBase functions in the advCalc code, we write the relevant value calculation LLVM codes. And we keep the result in a new register value. However, since the left rotate and right rotate functions do not have a direct equivalent in the IR language, we handle this case on 4 lines using subtraction, or , left and right shift.

```
fprintf(outputFile,"%%%d = add i32 0, %d\n", registerNumber, expression->value);
fprintf(outputFile,"%%%d = add i32 %%%d, %%%d\n", registerNumber, left_register, right_regis
```

5.5.4- Calling When we need to print a value, we print the result to the terminal using the relevant IR code and register number. In this part, we increase the register number by 1 and assign a number to the print part indirectly.

```
fprintf(outputFile,"%%%d = call i32 @printf(i8* getelementptr ([4 x i8], [4 x i8]* @.str, i3
```

6 - Improvements and Extensions

The operations that forced us to convert to llvm code or that we just tried were right rotate and left rotated. If possible, writing different operations using different logics would help us warm up and learn more about LLVM.

7 - Difficulties Encountered

Earlier in the advanced calculator homework we got a really good understanding of the structure of the homework by spending time. We understood and implemented the parse tree and the rest of the logic very well. For this assignment, we thought that converting the project we wrote in C to LLVM IR language would really force us, but we knew that we had to spend more time constructing the structure from the previous project about where and how we should do the load and store concepts we just saw, and we knew that we needed to create a good structure. Even though we spent time in the error logs and the newly added parts of the project, there was never a situation where we lost a lot of time and got stuck. ## 8 - Conclusion

We converted the advCalc language, which we previously wrote in C programming language, to LLVM IR code by writing a new transpiler. Building the structure and code well in the previous project helped us understand the logic and structure of this homework, and we had a pleasant project.

9 - Appendices

Source Code

CMPE 230 records and resources