CSC410 Project3 Final Report

Qiao Song Jiaqin Meng Yu-Hsuan Yang

The imperative-to-functional translation process

In order for us to properly process the translation of our inputs from an imperative language to a functional language, we started out by building our own abstract spanning tree to express our input imperative language. This involved dividing the input into individual categories that represented the different translation processes that each would have to go through. Since restrictions have been imposed on the let bindings that can appear in our final outputted functional language, we were able to restrict the number of nodes that we included in our AST. We ended up including the following nodes:

- ArrayRef: A node to translate array references or indexed in the code block
- BinaryOp: A node to translate binary operations (any mathematical operations with both a left and right side to the operation) in the code block
- Block: A node to process and store the code block itself
- Constant: A node to translate constant values in the code block
- ExprList: A node to translate the list of expressions that are assigned to each parent or node child in the functional output
- FileAST: A node to translate the code file as an AST
- FunctionDef: A node to translate function definitions in the code block
- ID: A node to identify the individual nodes
- IdentifierType: A node to store the type of each individual node
- ParamList: A node to store a list of the parameters included in a function definition
- TernaryOp: A node to store the condition and cases for if statements in the code block
- Let: A node to deal with assignments in the code block
- Letrec: A node to translate and process recursion in the code block

Next, we built a transform function to take the individual nodes built after processing the imperative language and visiting each node. This takes each individual case that may arise when visiting the following nodes and translates them.

- Block: Recursively transforms each item in the blocklist created when visiting the AST
- Assignment: Adds left side of assignment to return value, creates let statement for assignment, loops through following items
- BinaryOp: Transforms BinaryOp item, loops through following items
- Constant: Transforms Constant item, loops through following items
- If: Add required return items, loops through following items, if no else statement we will provide one tuple of written variable inside if.
- ID: Returns ID

- ArrayRef: Transforms ArrayRef item, loops through following items
- FuncCall: Tranforms FuncCall item, loops through arguments
- While: Checks condition, transforms loop items, loops until condition is met
- For: Checks conditions, transforms loop items, loops until condition is met
- Let or Letrec: Transforms Block items in block

After the imperative language has been fully translated, we simplify the outputted functional language to allow for easier reading and understanding. We do this in a few ways, first by checking to see if a variable on the left side of the operator is used in any operations below (by checking to see if it appears on the right side of the operation), disregarding the final output. If it used not used, we are able to replace it with the value assigned via the operation. Next, we check if a variable has been assigned a constant value only once. If so, we are able to replace the value wherever it appears in our block with the constant value given.

Unfortunately, we are not able to make “Simplify” work perfectly,because of structure changing after Simplify rule done and we do not have enough time to reconstruct “simplify” follow the new representing structure. We are still using the string comparing method with the output string.

Here is an working example:

```
fun block_function(a,b,c) returns (a, b, c)
=========transform ===========
Let a =  5  in
 Let b =  1  in
 Let c =  (b + 2)  in
 (a, b, c)
=========simplify===========
 Let c =  (1 + 2)  in
(5, 1, c)
```

The 2 basic rules mentioned by checkin 5 are still applied here.

How-to use tool

To Use our tool, there are some steps shown as follow.

Step 1: Cd into our project folder you cloned from our github repository
https://github.com/LittlePetunia/410p3

Step 2: Switch to the branch which has our final version of the project
submission_dec7

Step 3: Run in terminal by: >>>python transfunc.py
Step 4: You will see a message asking you for input followed by Step 3. Inputs are in a folder called “input”. Type input file name then you will get the output created by our program.(if run customer input, make sure put them into the input folder then do step 3 again, otherwise the file might not be found)

The syntax of functional language

Let...in…: Assignment statement in block of code

Letrec...in…: Loop name followed by condition followed by return value

Tuples: Return values for block of code

If...then...else…: If followed by condition followed by then followed by block of code with return value, then else followed by block of code with return value

Here is an example of if statement.

```
=======input=======
void test(){
if(b) {
      k = k + 1;
} else {
      k = k - 1;
}
c = a[k];
}
=============function==========
fun block_function(b,k,c,a) returns (k, c)
=========transform ===========
Let (k) =  if  b
 then
 Let k =  (k + 1)  in
 (k)
 else
 Let k =  (k - 1)  in
 (k)
  in
 Let c =  a[k]  in
 (k, c)
```