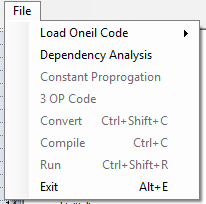
Michael Yahner  
Matt Farkas  
Optimization for Parallel Compilers Project Part II

**Program Overview**  
This second part of the programming project incorporates Three-Op transformation and Constant Propagation into our program. By modifying statements into Three-Op, It then makes the subsequent tasks of blocking, constant propagation, global expression replacement, and dead code removal easier to program to make a computer to do. There are many steps and specifics to consider when creating Three-Op code in our program, which is explained below.

We first had to consider, at what step in the whole optimization process to add the Three-Op code transformation in our program (i.e. Turn O’Neil code into Three-Op/ Turn C# code into Three-Op/ etc…). As there are many different paths to consider, it made sense to perform Three-Op conversion after dependency analyses and constant propagation, but before O’Neil code is converted to C# code.We controlled this in the GUI - that subsequent tasks must be completed, before Three-Op conversion could be ran. This can be seen in this following image of the File menu of our program below.



To accomplish Three-Op conversion, a class called “ThreeOpConverter” was created that transforms O’Neil code into 3OP O’Neil code. A method called “Transform” is called to start the conversion process. The header is below:

publicstaticstring[] Transform(List<string> code, List<ThreeOPCode>intStatements, List<ThreeOPCode>letStatements)

Note the following input parameters:

List<string> code: The list of strings which contain the untransformed O’Neil Code

List<ThreeOpCode>intStatements: An empty list of ThreeOpCreation type for int’s

List<ThreeOpCode>letStatements: An empty list of ThreeOpCreation type for let statements

You might be wondering what the ThreeOpCreation object used for. This is simply a class that was created to get and set index and statements of the temporary variables used in the Transform method.

The following Transform process is fairly complex, which includes a large amount of parsing based on different words, operations, and their relative locations in the statement. The following is a summarized flow of the process. For details, please review the ThreeOpConverter.cs class in the submitted code.

1. For Each Line of (O’Neil) code
   1. Make a copy of the original statement
   2. Trim the statement
2. If the first word is “let”, “print”, or “if”, then the line is a candidate for transformation.
3. Determine if the transform statements have multiple arrays and run the statement through one or more of the following functions:
   1. OneArrayTransformation: This function transforms ifs, lets, and prints/prompts arrays into Three-Op
   2. AfterEqualTransformation: Transforms statements that have multiple arrays
   3. ParenthesisTransformation: Transforms statements with parenthesis into Three-Op
   4. ValueExtraction: Creates a list of variables/constants in a non Three-Op statement
   5. OderOfOperations: Performs order of operations for all statements that already Three-Op
   6. LogicalOperator: Finds the logical operator in a statement
4. Finally, the results (the code that has been transformed into Three-Op) is returned and printed to the user in the GUI.

To get an idea of what type of line parsing is used in ThreeOpCreation, consider the following code snip-it in the OneArrayTransformation Method that parses text information from single-array statements (where “statement” is the current line of code being considered). Notice that positions of variable names, control flow syntax, and operations must be in their exact expected positions, or code will fail:

varindexBracketFront = statement.IndexOf("[", 0, StringComparison.Ordinal);

varindexBracketEnd = statement.IndexOf("]", 0, StringComparison.Ordinal);

. . .

if (int.TryParse(statement.Substring(indexBracketFront + 1, (indexBracketEnd - 1) - indexBracketFront), out result) == false)  
{  
 intStatements.Add(newThreeOPCode { Index = 2, Statement = "int t\_" + counter });  
 letStatements.Add(newThreeOPCode  
 {

Index = index,

Statement = original.Substring(0, original.IndexOf(type))   
 + "let " + "t\_" + counter + " = " +

statement.Substring(indexBracketFront + 1,

(indexBracketEnd - 1) - indexBracketFront)

});

Etc…

This next section will describe our attempt at constant propagation.

To assist in constant propagation, a class called “ConstantPropagation” was created. The method “Constants” is called to perform the propagation. Below is the function header, which simply takes the original code, and returns the code post constant propagation:

publicstaticstring[] Constants(List<string> code)

The following is a summarized flow of the constant propagation process. For details, please review the ConstantPropagation.cs class in the submitted code.

1. For each line of code
2. If the statement starts with “if” and contains “then” AND the statements contains a “let”, “goto”, “print”, or “prompt”, then continue
3. Find all variables created in the code and create a list by looking for let statements
4. Perform parsing on the lets statements to create the list of just variables
5. Remove variables from the list of variables that have more than one let statement - or they are used with an input which is a user accepted variable
6. For each variable, that are left
   1. These are the values that are constants
   2. For their let statement we take the value and add it to the list of that structure type to replace later
   3. The int statement is removed because the variable is a constant
7. For each variable, check the variables values against all the other variables to see if they are contained with in. meaning that if constant c is equal to constant b + 10 then the b in that statement needs to be replaced with its value
8. For each variable, This is where the values are actually replaced in the code

**Tests Performed**

A variety of tests were performed, including using all of the O’Neil test cases, and custom test cases to run our code. Unit testing was conducted when the code was being developed.

**Unexpected Behavior**

NOTES :

Automation - has a slight bug in that when it runs, anything other than 0 and 1 exit the program

Jacobi - still does not run because we still have not had to implement multi-dimensional arrays and Dr. O'Neil had talked about changing it just a one dimensional array

Sort Insertion - has a slight bug in, that when it runs, it does not swap variables correctly

Tax - crashes the program because a couple of the variables have not been initialized

All other cases worked fine