FINDING RELATIONSHIPS BETWEEN RENAMED AND REORDERED PROGRAMS

Le Chang
Le.Chang.GR@Dartmouth.edu

Follow this and additional works at: https://digitalcommons.dartmouth.edu/masters_theses

Recommended Citation
Chang, Le, "FINDING RELATIONSHIPS BETWEEN RENAMED AND REORDERED PROGRAMS" (2021).
Dartmouth College Master’s Theses. 45.
https://digitalcommons.dartmouth.edu/masters_theses/45

This Thesis (Master’s) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Master’s Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
FINDING RELATIONSHIPS BETWEEN RENAMED AND REORDERED PROGRAMS

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Master of Science
in
Computer Science
by
Le Chang
DARTMOUTH COLLEGE
Hanover, New Hampshire

Examinining Committee:

____________________________________
Sebastiaan Joosten, Chair

____________________________________
Douglas McIlroy

____________________________________
Sean Smith
Abstract

This paper describes a programming tool, DoRP, that can detect the differences between two LLVM programs. In particular, we wish to enable a comparison of the optimized and unoptimized output of GHC (Glasgow Haskell Compiler) at the LLVM stage. Because LLVM is a low-level language, programs written in this language are often large and hard to read. Moreover, the generated LLVM program is not only extensive but also has randomly named variables and randomly ordered functions, which increases its complexity. Thus, we have designed DoRP to help users learn the relationships between two LLVM programs quickly.

This tool is designed to compare two machine-generated LLVM programs. It can do the parameterized matching to find matched statements with or without substitutions of variable names, regardless of semantically insignificant reordering of these statements. Our method is parsing LLVM programs into a Haskell AST (abstract syntax tree) datatype, using these ASTs to build a graph, and finally finding the matched statements with a matching theory. DoRP can also reorder the input programs to improve the readability. The output is shown in a spreadsheet. Using the spreadsheet, users can directly see the similarity of structures of two programs and can adjust the output format. This paper discusses the method we used, experiments and results, and future work.
Preface

Throughout the writing of this thesis, I have received a lot of support and assistance. First and foremost, I would like to thank my advisor, Professor Sebastiaan Joosten, and Professor Douglas McIlroy, for their continuous support and encouragement. I could not finish this work without their immense knowledge and great experience. Their insightful feedback brought my work to a higher level. I am very fortunate to have them as my advisors. When I had problems, they were always there for help. I would also thank my committee member Professor Sean Smith for his kind help and support. Sincere thanks to Professor Eric Van Wyk for leading me to the beautiful world of programming language. A note of thanks to my friends for their support through tough times. Thanks, Zhehan and Simon, for bringing me this incredible spring. Final thanks to my parents and family for always being there to help me.
# Contents

Abstract ............................................................... ii
Preface ............................................................... iii

1 Introduction .................................................. 1

2 Related Work .................................................. 4

3 Method and Implementation .................................. 6
   3.1 Finding a Matching ....................................... 6
      3.1.1 Creating a Substitution Set ....................... 7
      3.1.2 Generating a Line Matching ...................... 10
   3.2 Generating an Output ................................... 12
      3.2.1 Checking Validation of Matching ................. 12
      3.2.2 Calculating a Heuristic Similarity Score ........ 13
      3.2.3 Reordering a Program ........................... 15
      3.2.4 Generating a Report ............................ 16

4 Experiments .................................................. 18

5 Discussion ..................................................... 21
   5.1 Future Work ........................................... 23

6 Conclusion ..................................................... 25
### References

#### Second Experiment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1 First Input Program</td>
<td>28</td>
</tr>
<tr>
<td>.2 Generated Files</td>
<td>34</td>
</tr>
<tr>
<td>.2.1 Variable Substitution</td>
<td>35</td>
</tr>
<tr>
<td>.2.2 Line Matching and Report</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Computer science is inseparable from human life today. We write code every day. Sometimes people want to check the back-end work of a compiler, but it takes people a lot of time and energy to read because of the large amount of code. Therefore, much work has focused on finding differences between large programs.

One way of finding differences is the character-based method [8, 5]. The most popular method is Unix Diff [8] that can find differences between two files line by line, using an LCS (longest common subsequence) algorithm. Other methods include using tree-to-tree-based [14, 3], graph-based [7, 6, 11, 4], meta-differencing [12], and etc. Most of these methods are designed for comparing text or finding duplication of human write programs, so they cannot handle renaming and reordering problems.

This report describes a programming language tool named DoRP (Differences of Randomly named/ordered Program). This language tool can compare two programs syntactically and semantically. Our original idea was to compare the optimization of the GHC compiler at the Haskell Core stage and LLVM stage. When looking at the LLVM code generated by GHC, we found that the generated LLVM program is much larger than expected. For example, one line "hello world" program compiled by clang would only generate about twenty lines LLVM program, but GHC compiles a one
line Haskell program, print "Hello world", and generates an LLVM program that has more than three hundred lines. What is more, a GHC-generated LLVM program has many variables with meaningless, randomly given names and may be present functions and attributes in random order. So even with two similar Haskell programs, GHC-generated LLVM programs would cause Diff to spot too many differences. Lots of those differences can be removed if we give the same use variable the same name. However, because the GHC-generated LLVM program is long, it requires much effort to find the matched variables in two programs. To find those variable matching and remove the differences that are not different after giving them the same variable names and statement sequences, we propose a method that combines a unification algorithm and a graph matching algorithm. This method can do parameterized match, i.e., it can match two different statements in character-based matching but are the same after substituting a statement’s variable names with corresponding names from the other statement.

Our method first parses the input program into an AST, then finds all the statements of functions (because functions are the immediate implementation of a program) and converts those statements into term lists. Next, it builds a weighted graph based on those two-term lists using a unification algorithm. Then it finds a maximum weighted matching of the built graph and uses this matching to detect the matched part of two input programs. The method also builds a constraints table for each program and checks the validation of the found matching with these two tables. The constraints table is built based on the LLVM program's semantics, and if there is any element against these constraints, our tool will show the warning message. Finally, it outputs a spreadsheet in CSV form. In this format, users can see matched statements and find their original position easily.

Compared to previous program comparison works, our method’s contribution is
that it can handle the renaming and reordering, so the differences it detects are more relevant and accurate. On the other side, because we use a constraints table to check the validation of found matchings, DoRP also considers the semantic correctness when finding differences of two programs.

The structure of this paper is: Section 2 discusses related work; Section 3 introduces details of our method and how we implement it in Haskell; Section 4 describes some experiments and their evaluation; Section 5 discusses the advantages and disadvantages of our method and future work; Section 6 is the conclusion.
Related Work

There were lots of works to find differences between the two programs. These works include character-based, tree-based, and statistical methods or combinations of these approaches. Here we briefly introduce some of those methods.

**Diff**  Diff is a UNIX utility\[8\]. It is text-based and used to compare text files. Diff uses LCS to detect text differences. Nevertheless, because it is not designed to compare two machine-generated programs, it cannot identify the randomly given names and random orders. The differences it finds are syntactic, and many of those differences are irrelevant.

**Statistical Method**  In 1988, H.T. Jankowitz designed an algorithm that can detect plagiarism in pascal programs\[9\]. This algorithm uses statistical analysis that first parses the program then constructs a static execution tree. With these trees, his algorithm can statistically analyze the particular styles like the use of operators and special symbols. This method can easily compare the similarity of two programs’ structures, but it is still hard to point out the exact location of differences.
Tree-Matching Method  Wuu Yang proposed a method to identify syntactic differences between two programs in 1991[14]. In this algorithm, two programs are firstly parsed into two ASTs, then do the tree matching. While comparing, a pretty printer would convert the visited tree nodes back to source code with differences highlighted. His method can filter out the irrelevant difference and match trees with different structures. However, Yang’s method cannot detect variable renaming.

Dup  Dup is a program implemented by Brenda Baker[2]. This program can detect occurrences of duplicated code in an extensive software system by building a suffix tree and then find the longest matching sections of code. The most exciting part is that Dup can do parameterized matching, which means non-declarative instances of names are canonicalized. It generates a report with variable substitutions. The problem is that Dup does not work with reordering.

Program Dependence Graphs  In 2001, J. Krinke proposed a method[11] that is based on fine-grained program dependence graphs, which are attributed to directed graphs. Such a graph can represent not only the structure but also the data flow of a program. The method identifies similar sub-graph structures by checking if two graphs are isomorphic. Still, this tool is not able to handle the renaming and reordering problem.

Each of those methods has advantages and limitations. Similar to those tree-based and graph-based methods, our tool requires that the input program is parsable. On the other hand, because we want to handle the renaming and reordering problem, instead of LCS, we use the unification algorithm and the graph matching algorithm to avoid comparing statements character-by-character.
Chapter 3

Method and Implementation

To compare two programs that have arbitrarily named variables and different orders of statements, one problem is how to find those variables and statements that are doing the same thing but have different names and orders while comparing. Therefore, we design this algorithm that can make a parameterized comparison. Our algorithm can be roughly divided into two parts: the first is to find matchings; the second is to generate a report on the relationships of two programs. It parses source code into an AST, uses the combination of a unification algorithm and a graph matching algorithm to find the substitution set for parameterized matching. Finally, it uses the found matching to calculate a similarity score and generate the output report.

Section 3.1

Finding a Matching

Finding matchings is one of the critical steps in our method. To learn the relationship between two programs, we need to know two different types of matchings: variable matchings and line matching. A variable matching, which we call a substitution set, stores the matchings of variables from one program to variables from the other program. We use variable matching to do line matching, which is used to keep paired
3.1 Finding a Matching Method and Implementation

3.1.1. Creating a Substitution Set

Because GHC generated programs have randomly named variables, it is crucial to figure out which variable can be substituted with an instance from the other program. If two variables have the same definitions and are used in identical situations, then we believe these two variables are paired and can be exchanged with each other. We create a substitution set to hold those paired variables. To form this set, we use the well-built Haskell LLVM parser called llvm-hs written by Benjamin S. Scarlet [1] to parse the LLVM program into the AST module. The AST module contains lots of information such as global attributes, global alias, etc., which are not very relevant to the functionality of a program. Moreover, because of the LLVM version problem, some of those attributes can be parsed into AST datatype successfully but can not be pretty-printed back into LLVM code. This is the problem of llvm-hs, and we cannot figure out how to fix it, but it harms finding line matching. Thus we remove those uninformative attributes and definitions and only keep the functions’ definitions.

Translating AST to Term List. The selected function definitions are stored in an AST data list, but we want to convert this function list into a term list because it has lots of information that we do not need for comparison. TermIndex is defined in Listing 1. Here TiVar, TiConst, and TiConstTy are constructors for constant values like variables names, global/local references, integers and etc. TiApp means application. It can be used for keeping either sequences of LLVM commands or functions such as store, alloca, load, etc. We also give TiApp an index i for identification. AppFunction is in App is defined to show type of stored information. Seq is for sequence of statements; UserDefined represents this function is not a build-in LLVM command but defined by user (compiler); Other is just the opposite of UserDefined, it is for built-in LLVM
3.1 Finding a Matching Method and Implementation

```
data TermIndex i = TiVar LLVM.AST.Name
     | TiConst LLVM.AST.Operand
     | TiConstTy LLVM.AST.Type
     | TiApp i AppFunction [TermIndex i] (Maybe OriginalAST)
     ↦ deriving (Eq, Show, Functor, Traversable, Foldable, Read)

data OriginalAST = Ins (LLVM.AST.Named LLVM.AST.Instruction)
     | Tem (LLVM.AST.Named LLVM.AST.Terminator)
     deriving (Eq, Show, Read)

data AppFunction = Seq
     | UserDefined LLVM.AST.Name
     | Other LLVM.AST.Name
     | Arguments Int deriving (Eq, Show, Read)
```

Listing 1: Haskell Definition of Term

commands; Arguments means this App has the information of a function’s arguments. Maybe OriginalAST is designed to hold original AST data which will be used later when finding line matching. OriginalAST has two types: Named Instruction and Named Terminator. These two types are the only two types that occur in functions. Instruction has type like Store, Alloca, Load... which are the common commands of LLVM and Terminator indicates blocks that should be executed next, which includes Ret, CondBr... An example of how to convert a LLVM command into Term is shown in Listing 2. Here LLVM code: store i32 0, i32 %y1, align 4 is parsed into AST datatype like shown at line 5. Then we translate this AST into TermIndex at line 8.

**Building a Graph and Finding a Graph Matching.** With term lists of input source code, we can move on to the next step: building the substitution set. The main idea on how to build this set is to create a graph and find the matching. First, we take an element from the first term list and run unification on this element with every element in the second list. The unification algorithm will return a substitution if two terms can be unified successfully. Our program does this step to all elements
in the first list and collects all the generated substitutions, combining them into a single list A. Because a substitution is a set of paired variables and terms, we then treat every unique term in list A as a vertex. If there is a mapping between a pair of vertices, we add an edge between these two vertices. In this way, we can build a graph of terms. An example of how we build a bipartite graph is shown in Figure 3.1.

This graph is bipartite because based on the way we build the graph, no edge will be added between terms from the same program. Therefore, these vertices can be divided into two sets, U and V, where U has all terms from the first program and V has all terms from the second program. Since every term in U can be matched with several terms in V, every edge we add connects a vertex in U to one in V. From this bipartite graph, we can find many matchings. These matchings are the potential substitution set of terms. We decide to find a matching because every variable can be substituted with one term only, and matching satisfies this requirement. Because
3.1 Finding a Matching and Implementation

Figure 3.1: Example of Building Bipartite Graph and Find Maximum Weighted Matching

It has more than one matching, we need to choose one among them. To do that, we improve our method by adding weight to edges. The weight is the number of times a paired variable and term occur. In this way, we are able to find the matching that has the highest frequency. In other words, for a variable that has more than one matched term, we pick the term with the most count of pairing. The algorithm for finding maximum weighted matching we use is the Hungarian method [10].

3.1.2. Generating a Line Matching

We now can find the line matching with the variable matching. A line matching is a list of tuples of line index of source code. It shows the matching lines of the input LLVM program directly. Our approach of generating line matching is that we first substitute all variable names in the first program with related terms from the second program based on the previously created substitution set. We assign the substituted term list to a variable named renamed-P1. Then we compare a term, a, in renamed-P1 to all terms from the second term list. The first equal term that has not been matched
before is selected to be paired with a. We get the index of this pair of terms and store those two indexes as a tuple. By mapping this step to all terms from the first term list, we can get a list of tuples of integer, which is a term matching.

A term matching is not intuitive because the term index is different from their line index in the input program. Hence, we should find the corresponding line index of those paired terms and convert this term matching into a line matching. Here is where the Maybe OriginalAST is designed for. In the beginning, our idea was to write a transformation function that transforms TermIndex back to Haskell AST datatype. Soon we found that when converting AST to TermIndex, there was lots of information lost because we do not need them for comparison. Therefore, we decide to add a field called OriginalAST to the TermIndex to hold the Haskell AST value of a term. When comparing the equivalence of two terms, OriginalAST is not considered. The reason OriginalAST is a Maybe type is that some terms like block number do not need to store their original AST, so the value of those terms’ OriginalAST field is Nothing.

Having this OriginalAST, we can translate TermIndex back to LLVM source code and use string matching to find the matched lines in the original program. We utilize the pretty-printer of Haskell AST to get the LLVM code of Haskell AST data. The two LLVM source programs are read as two strings and are split into two string lists by ”\n”. Same as how we find the term matching, we take a tuple from term matching, looking up their indexes to get the TermIndex and extract their OriginalAST. Then compare the pretty-printed LLVM code string with all strings in the corresponding string-list. The first found equal strings, which the indexes have not been matched before, are used to form a paired line number tuple. Our program does this step to all elements in the term matching to get the line index matching.
Generating an Output

Previously generated variable substitution and line matching are stored in two separate text files. In generating output, DoRP takes in those two text files and two LLVM source programs. It first checks if the take-in matchings are semantically correct, and then we can generate the report showing how two programs are related to each other, what are their similarity, and what their differences are. To quantify the similarity, we let DoRP calculate a heuristic score. This score indicates how many terms are matched between two programs. We also generated a report to visualize the matched part of two programs. Therefore, our output mainly has four steps: check if the input matchings are valid, calculate a heuristic similarity score, an optional step of reordering program, and generate the spreadsheet format file.

3.2.1. Checking Validation of Matching

Before we calculate the score, we need to check if the matching is valid. For example, in Listing 3, the generated variable matching is \([(x1,x2),(y1,y2)]\) and line matching is \([(1,1), (2,2), (3,4), (4,3)]\). However, in program1, x1 is given a new value after calling function f, and in program2, x2 is given a new value first, then be used in f. So even though the generated matching is correct, it might not be proper semantically.

To check this error, we build a constraint table for each program inspired by Yui Sasaki et al. [13]. This constraint table has four columns: 1. a term that has variable; 2. a list of terms that redefine the variables in the first column (this column is named defDef); 3. a list of terms that uses variables that occur in the first column (we called this column defUse); 4. a column that has all terms that occur after the term in the first column if that term is a terminator (this column is named escape). This constraint table defines some fixed order that should not be reversed. We used these
3.2 Generating an Output Method and Implementation

Listing 3: Example of Invalid Matching

---

3.2.2. Calculating a Heuristic Similarity Score

One of our goal is to compare the two LLVM programs and find their similarity and differences. To quantify the similarity, we propose a function called `calculateScore`
3.2 Generating an Output Method and Implementation

--program1 --program2

x1 = 2  y2 = 2
y1 = 2  x2 = 2
z1 = 3  z2 = 3

--matching
[(x1,y2),(y1,x2),(z1,z2)] -- variable matching
[(1,2), (2,1), (3,3)] -- line matching

Listing 4: Example of a Valid Matching but not what user expect

that takes two term lists, a line matching and a substitution. It returns an integer as a
heuristic score of similarity. The higher the score, the more similar the two programs
are. The definition of calculateScore is shown below:

\[
\text{calculateScore}(\text{terms}_1, \text{terms}_2, \text{matching}, \text{subst}) = \begin{cases} 
0 & \text{if scores has 0} \\
\sum_{i=0}^{n} \text{scores} & \text{otherwise}
\end{cases}
\]

where \( \text{scores} = \text{map}(\text{score}(\text{terms}_1, \text{terms}_2, \text{subst})) \text{matching} \)

From the definition, calculateScore takes four arguments: two term lists, a term
matching and a substitution set. It passes these four arguments to a function score.
The definition of score is shown here:

\[
\text{score}(\text{terms}_1, \text{terms}_2, (a, b), \text{subst}) = \begin{cases} 
1 & \text{if terms}_1[a] = \text{terms}_2[b] \\
0 & \text{otherwise}
\end{cases}
\]

Function score returns 0 if two matched terms are not equal to each other and 1
otherwise. If the score is 0, that means there exists an invalid pair of terms in the
input matching. By mapping this function to the term matching list, we can get a list
of scores. Then in calculateScore, we get the sum of the score list and the product
of the score list. The sum is the maximum score calculateScore can return, but
instead of letting it return the sum directly, we multiply the sum and product of the score list, so if there is any invalid matching exists, the return score is zero.

To make this score more meaningful, we want to know an upper bound of the score. The method is first rename all variables in two programs canonically, and then for each line in the first program, if there is an equal line found in the second program, the score is 1. The sum of scores of lines in the first program is the upper bound, i.e., the maximum value our calculateScore can get. Comparing the upper bound with the calculated score can tell the user how many lines are matched between two input programs and how many lines are not. So users can have a general idea of how many statements in these two programs are equivalence and how many lines are different.

3.2.3. Reordering a Program

We find that GHC generated code is hard to compare by humans because the distances of some matched lines in two programs are far from seeing. Therefore, we want to add functionality to DoRP which can reorder the program to help shorten the distance between matched lines. The foundation of this idea is from [13]. Based on the constraint table built before, we try to get all the permutations of lines that do not violate those constraints. Then we implement a function to calculate the distance of these permutations called calculateDistance. This function takes one permutation from each program and the generated line matching as input. From the first permutation, if a line $l_1$ has a matched line $l_2$ in matching, then find the index of $l_2$ in the second position. The subtraction of the index of $l_1$ and the index of $l_2$ is the distance of these two matched lines. Repeating this step to all lines in the first permutation, we get a distance list, and the sum of this list is the final distance that calculateDistance returns. DoRP chooses two permutations with the smallest distance value as the reordered program to be outputted in the final report. An
example of a reordering program is shown in Figure 3.2. In this example, the upper left is the two input programs, and the distance of matched lines is 4. The lower are two programs after reordering (the second program is reordered). The distance after reordering the second program is 0. So the two programs become more similar.

| 1 x1 = 1 | 1 z2 = 3 | Distance: |
| 2 y1 = 2 | 2 y2 = 2 | (x1,y2) -> |1-3| = 2 |
| 3 z1 = 3 | 3 x2 = 1 | (y1,y2) -> |2-2| = 0 |
|           |           | (z1,z2) -> |3-1| = 2 |
|           |           | Total: 4   |

| 1 x1 = 1 | 1 x2 = 1 | Distance: |
| 2 y1 = 2 | 2 y2 = 2 | (x1,x2) -> |1-1| = 0 |
| 3 z1 = 3 | 3 z2 = 3 | (y1,y2) -> |2-2| = 0 |
|           |           | (z1,z2) -> |3-3| = 0 |
|           |           | Total: 0   |

Figure 3.2: Example of Reordering Program

### 3.2.4. Generating a Report

The final step of our tool is to visualize the output by generating a comparison report. We choose to use the spreadsheet as our report format. The generated similarity report has at least six columns which are: the line number of code, the source code of the first program, the lines matched with the first program from the second program, the line index of matched statements from the second program, the line index of original of the second program and the source code of the second program. We think a spreadsheet is intuitive for users because it is easy to see the index, lines, and matched lines in the same row, and users can adjust the order of lines by switching the whole row quickly. Before generating the final report, our tool will ask the user if they want to see the program after reordering. If the user answers yes, then the report
3.2 Generating an Output Method and Implementation

will have four extra columns, which are the reordered source LLVM program and the reordered original line indexes. The report is stored in a CSV file, and columns are separated by the question mark (because this is the symbol that does not occur in the regular LLVM program). We first count how many lines each program has, then create two lists to store the index, then we find the matching lines and their indexes and store them in other lists. Finally, we have at least six lists, and we write a zip function to zip them into a single list and convert this list into a string in CSV format.
To check its functionality, we test this tool with several experiments. Each experiment has two input programs that may or may not have the same length. We evaluate the result from three aspects: generated report correctness, the heuristic score, and the runtime.

The first experiment compares two LLVM programs generated by Clang because Clang generated code is much smaller than GHC. The two input programs have seven lines. Note that to show the differences clearly, we manually rename the variables in two programs. Five of these lines are expected to be matched. The generated report is shown in Figure 4.1. This report shows the reordering of two programs. Column E shows the matched part of two programs which is, as expected, five lines. The score of this function is 7. This is because except those five lines, function main and block #0 at line 6 are parsed as two terms, and they are matched. However, these two terms are not inside the function definitions, so they are not shown in Column E. It takes less than 1 second to generate this output. So we think DoRP works well on small examples.
For the second experiment, we use GHC generated LLVM programs as input. Because GHC is nondeterministic, we can compile a one-line Haskell program:

```haskell
main = print 0
```

twice to get two different LLVM programs. These two generated LLVM programs have 317 lines. Even though the order of statements is the same, they have lots of randomly given names, and if we run Diff to compare them, it will have about three hundred differences. The first time we give DoRP these two programs directly, the report shows there are around two hundred and fifty lines matched. The heuristic score is 259, and the upper bound is 259 as well, which means most lines can be matched, and these two programs are very similar, just as we expect. We also try to reorder some part of the code and retest it manually. The result is just the same as the first time, which proves that our tool can handle the randomly ordering program. The runtime of these three-hundred lines is about 1 minute. We think this is also acceptable. The input LLVM program and generated files are put in appendix A.

The third experiment is with two more complex GHC generated LLVM programs with more than two thousand lines. The generated report shows about five hundred
lines matched, and the score is about the same number. This result is not as good as what we expect, as we believe there should be more than this matched. One problem we can find is when finding the maximum matching for substitution set, there is more than one choice, and the Hungarian method chooses the first one. This chose matching is different from the substitution found by a human. The other problem this experiment shows is the runtime. For example, it takes more than five hours on a laptop computer to complete the finding matchings steps. Because of a large amount of code takes about an hour to generate the graph and even more time to find the matching and generate the substitution. This means our tool has poor efficiency when run on large programs.

For all the experiments, we record the time spent for generating graphs, finding variable substitutions and matchings, and checking validation in a table which is shown in Table 4.1.

<table>
<thead>
<tr>
<th>Line number of input programs (fst, snd)</th>
<th>Building Graph (seconds)</th>
<th>Finding Substitution(s)</th>
<th>Generating Matching(s)</th>
<th>Checking(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(23,23)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(121,121)</td>
<td>0.01</td>
<td>0.05</td>
<td>0</td>
<td>0.3</td>
</tr>
<tr>
<td>(317,317)</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>(446,446)</td>
<td>7</td>
<td>10</td>
<td>0.01</td>
<td>11</td>
</tr>
<tr>
<td>(870,870)</td>
<td>152</td>
<td>399</td>
<td>0.1</td>
<td>473</td>
</tr>
<tr>
<td>(1151,1151)</td>
<td>468</td>
<td>1440</td>
<td>1</td>
<td>1627</td>
</tr>
<tr>
<td>(1321,1321)</td>
<td>1245</td>
<td>4636</td>
<td>1</td>
<td>4844</td>
</tr>
<tr>
<td>(2056,2056)</td>
<td>5220</td>
<td>10002</td>
<td>1</td>
<td>9990</td>
</tr>
</tbody>
</table>

Table 4.1: Table to test captions and labels
We developed a tool to compare the difference between two LLVM programs. It is entirely written in Haskell. After testing, we believe our tool can provide helpful information for users to learn the differences and similarities between the two programs instantly. The substitution set can do parameterized matching, ignoring the different variable names in two programs while comparing and ignoring the different order of statements. It can point out the differences accurately. At the same time, the generated report can be customized by user option to provide more information, including reordering the programming to increase readability. While DoRP has these functionalities, our tool still has several drawbacks.

First, the input program needs to be syntactically correct to be parsed into AST. For our intended use of machine-generated LLVM code, incorrect input is unlikely. However, to extend the usage of our tool to human written programs, it is necessary to consider the incomplete program.

Second, the tool has low efficiency. Based on Table 4.1, we create a chart that is shown in Figure 5.1. We are not able to given an accurate runtime equation because the runtime of some built method like the Hungarian method, permutation method is not provided, but from the semi-log graph, we know that it is faster than exponential.
From the log-log graph, we can see the estimated runtime could possibly be $O(N^4)$, which is slow when $n$ is large. Our experiments show that when the input program has more than one thousand lines, the time spent on building a graph, matching and checking becomes large. We can improve our algorithm to make it run faster.

Third, there exists information lost when parsing LLVM code into AST, such as attributes definitions, function declarations. This would cause us to miss some differences or matched statements. For example, the LLVM source code has the value of an address of a pointer; after parsing, this address value lost, and when converting the parsed AST back to LLVM, the address could not be found. In addition, there
are some function attributes that can be parsed into AST but cannot be converted back in LLVM code. Right now, DoRP cannot handle it because these problems are not from our algorithm but from those libraries we use. It is hard to figure out where and why these happen and how to fix them.

Fourth, because we use the Hungarian method from the Haskell package to find the maximum weighted matching, this function may choose the first option if there is more than one matching. (We are not sure about this because we have not to look at the source code of this function, but based on its performance, our assumption is it chooses the first matching it can find.) However, there are many duplicated lines with only variable name differences. This choice made by the Hungarian method might be different from human write substitution. It would be great to write our algorithm to let it choose the matching based on some constraints found by the machine or provided by the user.

Section 5.1

Future Work

Our tool still has a lot of improvement space. In the future, we will keep working on developing it. There are several things we think that we can improve to make DoRP functional.

- Making DoRP handle more LLVM commands. The current tool can only handle some LLVM commands frequently used, but LLVM has many instructions. So if the input program is very complex, our tool would not be able to manage it. It is necessary to finish this part to make it workability complete.

- Improving efficiency. $O(N^4)$ is a prolonged runtime. To improve it, we should revisit our algorithm. One way to increase the runtime efficiency is to break
large LLVM programs into several small blocks, based on semantical analysis, and compare these small blocks first. Because blocks have fewer lines than the whole program, the runtime might be faster than what we have right now. In that way, we might be able to improve it to $O(N^2)$.

- Adding more options. Because this tool is designed to help the user learn the relationship of two extensive LLVM programs easily, adding options like letting users input constraints for finding substitution sets and line indexes that restrict the range of lines for comparing can help improve users’ experience.

- Improving output. For the output of the reordered program, expect showing it in the spreadsheet report, we also create a function that can write the reordered program into a .myHs format, and we develop a highlighter for .myHs format to indicate which lines’ position are changed where is their original position. Although the spreadsheet can show the same information, we think highlighting can tell the user the differences more intuitively. However, this highlighter does not work for the spreadsheet format, so we hope we could add this to our output in the future.
Chapter 6

Conclusion

In this paper, we introduce a tool DoRP that can be used to compare and improve the readability of two programs. LLVM is an Intermediate Representation that the compiler can generate. The compiler-generated IR can be hard to read due to high abstraction and unpredictable compiler optimization. To find the relationship between two different machine-generated LLVM programs with randomly given names and randomly ordered statements, we propose a method that combines a graph matching algorithm and a unification algorithm to do the parameterized matching. This tool can generate a report in spreadsheet format to show the matched parts of programs and calculate a heuristic score to tell the user how similar these two input programs could be. DoRP can handle most LLVM command and relatively small programs. When facing programs with more than a thousand lines, it takes a very long time, and the result is not as good as expected. In the future, we want to improve our algorithm to make our tool more efficient and can generate the better report.
Bibliography


Second Experiment

Section .1

First Input Program

target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux"
declare ccc i8* @memcpy$def(i8*, i8*, i64)
declare ccc i8* @memmove$def(i8*, i8*, i64)
declare ccc i8* @memset$def(i8*, i64, i64)
declare ccc i64 @newSpark$def(i8*, i8*)
!0 = !{"root"}
!1 = !{"top", !0}
!2 = !{"stack", !1}
!3 = !{"heap", !1}
!4 = !{"rx", !3}
!5 = !{"base", !1}

@sPk_closure_struct = type <{i64, i64}>
@sPk_closure$def = internal global %sPk_closure_struct<{i64 ptrtoint (i8* @integerzmwiredzmin_GHCziIntegerziType_Szh_con_info to i64), i64 1}>
@sPk_closure = internal alias i8, bitcast (%sPk_closure_struct* to i8*)

%_uPu_srt_struct = type <{i64, i64, i64, i64}>
%_uPu_srt$def = internal global %_uPu_srt_struct<{i64 ptrtoint (i8* @stg_SRT_3_info to i64), i64 ptrtoint (i8* @base_SYSTEMziIO_print_closure to i64), i64 ptrtoint (i8* @base_GCHCziShow_zdfShowInteger_closure to i64), i64 0}>
_uPu_srt = internal alias i8, bitcast (%_uPu_srt_struct* to i8*)

%Main_main_closure_struct = type <{i64, i64, i64, i64}>
@Main_main_closure$def = internal global %Main_main_closure_struct<{i64 ptrtoint (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @Main_main_info$def to i64), i64 0, i64 0, i64 0}>
@Main_main_closure = alias i8, bitcast (%Main_main_closure_struct* to i8*)
@Main_main_info = alias i8, bitcast (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @Main_main_info$def to i8*)
define ghccc void @Main_main_info$def(i64* noalias nocapture %Base_Arg, i64* noalias nocapture %Sp_Arg, i64* noalias nocapture %Hp_Arg, i64 %R1_Arg, i64 %R2_Arg, i64 %R3_Arg, i64 %R4_Arg, i64 %R5_Arg, i64 %R6_Arg, i64 %SpLim_Arg) align 8 nounwind prefix <{i64, i32, i32}><{i64 0, i32 21, i32 add (i32 trunc (i64 sub (i64 ptrtoint (%_uPu_srt_struct* @_uPu_srt$def to i64),i64 ptrtoint (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @Main_main_info$def to i64)) to i32),i32 0})>

{nPv:
  %lrgc = alloca i64, i32 1
  %R3_Var = alloca i64, i32 1
  store i64 undef, i64* %R3_Var
  %R4_Var = alloca i64, i32 1
  store i64 undef, i64* %R4_Var
  %R5_Var = alloca i64, i32 1
  store i64 undef, i64* %R5_Var
  %R6_Var = alloca i64, i32 1
  store i64 undef, i64* %R6_Var
  %F1_Var = alloca float, i32 1
  store float undef, float* %F1_Var
  %D1_Var = alloca double, i32 1
  store double undef, double* %D1_Var
  %F2_Var = alloca float, i32 1
  store float undef, float* %F2_Var
  %D2_Var = alloca double, i32 1
  store double undef, double* %D2_Var
  %F3_Var = alloca float, i32 1
  store float undef, float* %F3_Var
  %D3_Var = alloca double, i32 1
  store double undef, double* %D3_Var
  %F4_Var = alloca float, i32 1
  store float undef, float* %F4_Var
  %D4_Var = alloca double, i32 1
  store double undef, double* %D4_Var
  %F5_Var = alloca float, i32 1
  store float undef, float* %F5_Var
  %D5_Var = alloca double, i32 1
  store double undef, double* %D5_Var
  %F6_Var = alloca float, i32 1
  store float undef, float* %F6_Var
  %D6_Var = alloca double, i32 1
  store double undef, double* %D6_Var
  %lcPo = alloca i64, i32 1
  %R2_Var = alloca i64, i32 1
  store i64 undef, i64* %R2_Var
  %R1_Var = alloca i64, i32 1
  store i64 %R1_Arg, i64* %R1_Var
  %Sp_Var = alloca i64*, i32 1
  store i64* %Sp_Arg, i64** %Sp_Var
  br label %cPr

  %lnPw = load i64, i64* %R1_Var
  store i64 %lnPw, i64* %lrgc
  %lnPx = load i64, i64* %Sp_Var
  store i64 %lnPx, i64* %lnPw
  %lnPy = getelementptr inbounds i64, i64* %lnPx, i32 1
  %lnPz = ptrtoint i64* %lnPy to i64
  %lnPA = sub i64 %lnPz, 24
  %lnPB = icmp ult i64 %lnPA, %lnPA, %SpLim_Arg
  %lnPD = call ccc i1 (i1, i1) @llvm.expect.i1( i1 %lnPB, i1 0 )
  br i1 %lnPD, label %cPs, label %cPt

  %lnPE = ptrtoint i64* %Base_Arg to i64
  %lnPF = inttoptr i64 %lnPE to i8*
Second Experiment

\%lnPG = load i64, i64* %lrgc
\%lnPH = inttoptr i64 %lnPG to i8*
\%lnPI = bitcast i8* @newCAF to i8* (i8*, i8*)
store i64 undef, i64* %R3_Var
store i64 undef, i64* %R4_Var
store i64 undef, i64* %R5_Var
store i64 undef, i64* %R6_Var
store float undef, float* %F1_Var
store double undef, double* %D1_Var
store double undef, double* %D2_Var
store double undef, double* %D3_Var
store double undef, float* %F2_Var
store double undef, float* %F3_Var
store double undef, double* %D4_Var
store double undef, float* %F4_Var
store double undef, double* %D5_Var
store double undef, float* %F5_Var
store double undef, float* %F6_Var
store double undef, double* %D6_Var
\%lnPJ = call ccc i8* (i8*, i8*) %lnPI( i8* %lnPF, i8* %lnPH ) nounwind
\%lnPK = ptrtoint i8* %lnPJ to i64
store i64 %lnPK, i64* %lcPo
\%lnPL = load i64, i64* %lcPo
\%lnPM = icmp eq i64 %lnPL, 0
br i1 %lnPM, label %cPq, label %cPp

\%lnPO = ptrtoint i8* @stg_bh_upd_frame_info to i64
\%lnPN = load i64*, i64** %Sp_Var
\%lnPP = getelementptr inbounds i64, i64* %lnPN, i32 -2
store i64 %lnPO, i64* %lnPP, !tbaa !2
\%lnPR = load i64, i64* %lcPo
\%lnPS = load i64*, i64** %Sp_Var
\%lnPS = getelementptr inbounds i64, i64* %lnPS, i32 -1
store i64 %lnPR, i64* %lnPS, !tbaa !2
\%lnPT = ptrtoint %sPk_closure_struct* @sPk_closure$def to i64
\%lnPU = add i64 %lnPT, 1
store i64 %lnPU, i64* %R3_Var
\%lnPV = ptrtoint i8* @base_GHCziShow_zdfShowInteger_closure to i64
store i64 %lnPV, i64* %R2_Var
\%lnPv = ptrtoint %sPk_closure_struct* @sPk_closure$def to i64
store i64 %lnPv, i64* %R1_Var
\%lnPX = load i64*, i64** %Sp_Var
\%lnPY = getelementptr inbounds i64, i64* %lnPX, i32 -2
\%lnPz = ptrtoint i64* %lnPY to i64
\%lnQ0 = inttoptr i64 %lnPz to void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64, i64)*
\%lnQ1 = bitcast i8* @stg_ap_pp_fast to void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64, i64)*
\%lnQ2 = load i64, i64* %lrgc
\%lnQ3 = load i64, i64* %lrgc
\%lnQ4 = load i64, i64* %lrgc
\%lnQ5 = load i64, i64* %lrgc
tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64, i64)*
\%lnQ6( i64* %Base_Arg, i64* %Hp_Arg, i64 %lnQ2, i64* %lnQ3, i64 %lnQ4,
\%lnQ5, i64 undef, i64 undef, i64 undef, i64 %SpLim_Arg ) nounwind
ret void

\%lnQ6 = load i64, i64* %lrgc
\%lnQ7 = inttoptr i64 %lnQ6 to i64*
\%lnQ8 = load i64, i64* %lnQ7, !tbaa !1
\%lnQ9 = inttoptr i64 %lnQ8 to void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64, i64)*
\%lnQa = load i64*, i64** %Sp_Var
Second Experiment

\[
\%lnQb = \text{load } i64, i64* \%R1_Var
\]
\[\text{tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)}\]
\[
\%lnQ9( \text{i64* }%Base_Arg, \text{i64* }%lnQa, \text{i64* }%Hp_Arg, \text{i64 }%lnQb, \text{i64 undef, i64 undef, i64 undef, i64 undef, i64 }%SpLim_Arg ) \text{ nounwind}
\]
\[\text{ret void}\]
\[
\%
\]
\[\%lnQg = \text{load } i64, i64* \%R1_Var
\]
\[\text{store i64 }%lnQc, \text{i64* }%R1_Var
\]
\[\%lnQd = \text{getelementptr inbounds i64, i64* }%Base_Arg, \text{i32 }-2
\]
\[\%lnQe = \text{bitcast i64* }%lnQd \text{ to } i64*
\]
\[\%lnQf = \text{load i64, i64* }%lnQe, !\text{tbaa }5
\]
\[\%lnQg = \text{inttoptr i64 }%lnQf \text{ to void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)}
\]
\[\%lnQh = \text{load i64*, i64* }%R1_Var
\]
\[\%lnQi = \text{load i64, i64* }%R1_Var
\]
\[\text{tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)}\]
\[
\%lnQg( \text{i64* }%Base_Arg, \text{i64* }%lnQh, \text{i64* }%Hp_Arg, \text{i64 }%lnQi, \text{i64 undef, i64 undef, i64 undef, i64 undef, i64 }%SpLim_Arg ) \text{ nounwind}
\]
\[\text{ret void}\]
\]
\[\text{declare ccc i1 }@\text{llvm.expect.i1(i1, i1)}\]
\[\%_uQs_srt_struct = \text{type }\{i64, i64, i64, i64\}
\]
\[\%ZCMain_main_closure_struct = \text{type }\{i64, i64, i64, i64\}
\]
\[@\text{uQs_srt$def} = \text{internal global }%_uQs_srt_struct\{\text{i64 ptrtoint (i8* @stg_SRT_2_info to i64), i64 ptrtoint (i8* @base_GHCziTopHandler_runMainIO_closure to i64), i64 ptrtoint to i64), i64 0}\}
\]
\[@\text{uQs_srt} = \text{internal alias i8, bitcast }(%_uQs_srt_struct* @_uQs_srt$def to i8*)
\]
\[@\text{ZCMain_main_closure$def} = \text{internal global }%ZCMain_main_closure_struct\{\text{i64 ptrtoint (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @ZCMain_main_info$def to i64), i64 0}\}
\]
\[@\text{ZCMain_main_closure} = \text{alias i8, bitcast }(%ZCMain_main_closure_struct*)
\]
\[@\text{ZCMain_main_info$def} = \text{internal global }%ZCMain_main_info_struct\{\text{i64, i64, i64, i64, i64, i64, i64, i64, i64, i64}\}
\]
\[@\text{ZCMain_main_info} = \text{alias i8, bitcast }(%ZCMain_main_info_struct*)
\]
\[\text{define ghccc void }@\text{ZCMain_main_info$def}(\text{i64* }\text{noalias nocapture }%Base_Arg, \text{i64* }\text{noalias nocapture }%Sp_Arg, \text{i64* }\text{noalias nocapture }%Hp_Arg, \text{i64 }%R1_Arg,
\]
\[\text{i64 }%R2_Arg, \text{i64 }%R3_Arg, \text{i64 }%R4_Arg, \text{i64 }%R5_Arg, \text{i64 }%R6_Arg, \text{i64 }%SpLim_Arg) \text{ align 8 nounwind prefix }\{\text{i64, i32, i32}\}<\text{i64 0, i32 21, i32 132}
\]
\[\text{132 truncate (i64 sub (i64 ptrtoint }(%_uQs_srt_struct* @uQs_srt$def to i64),i64 ptrtoint (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @ZCMain_main_info$def to i64)) to i32),i32 0)\}
\]
\[\}
\]
\[\text{nQt:}
\]
\[\%l01D = \text{alloca i64, i32 1}
\]
\[\%R3_Var = \text{alloca i64, i32 1}
\]
\[\text{store i64 undef, i64* }\%R3_Var
\]
\[\%R4_Var = \text{alloca i64, i32 1}
\]
\[\text{store i64 undef, i64* }\%R4_Var
\]
\[\%R5_Var = \text{alloca i64, i32 1}
\]
\[\text{store i64 undef, i64* }\%R5_Var
\]
\[\%R6_Var = \text{alloca i64, i32 1}
\]
\[\text{store i64 undef, i64* }\%R6_Var
\]
\[\%F1_Var = \text{alloca float, i32 1}
\]
\[\text{store float undef, float* }\%F1_Var
\]
\[\%D1_Var = \text{alloca double, i32 1}
\]
\[\text{store double undef, double* }\%D1_Var
\]
\[\%F2_Var = \text{alloca float, i32 1}
\]
\[\text{store float undef, float* }\%F2_Var
\]
\[\%D2_Var = \text{alloca double, i32 1}
\]
\[\text{store double undef, double* }\%D2_Var
\]
\[\%F3_Var = \text{alloca float, i32 1}
\]
store float undef, float* %F3_Var
%D3_Var = alloca double, i32 1
store double undef, double* %D3_Var
%F4_Var = alloca float, i32 1
store float undef, float* %F4_Var
%D4_Var = alloca double, i32 1
store double undef, double* %D4_Var
%F5_Var = alloca float, i32 1
store float undef, float* %F5_Var
%D5_Var = alloca double, i32 1
store double undef, double* %D5_Var
%F6_Var = alloca float, i32 1
store float undef, float* %F6_Var
%D6_Var = alloca double, i32 1
store double undef, double* %D6_Var
%lcQm = alloca i64, i32 1
%R2_Var = alloca i64, i32 1
store i64 undef, i64* %R2_Var
%R1_Var = alloca i64, i32 1
store i64 %R1_Arg, i64* %R1_Var
%Sp_Var = alloca i64*, i32 1
store i64* %Sp_Arg, i64** %Sp_Var
br label %cQp

cQp:
%lnQu = load i64, i64* %R1_Var
store i64 %lnQu, i64* %l01D
%lnQv = load i64*, i64** %Sp_Var
%lnQw = getelementptr inbounds i64, i64* %lnQv, i32 1
%lnQx = ptrtoint i64* %lnQw to i64
%lnQy = sub i64 %lnQx, 24
%lnQz = icmp ult i64 %lnQy, %SpLim_Arg
%lnQA = call ccc i1 (i1, i1) @llvm.expect.i1( i1 %lnQz, i1 0 )
br %i1 %lnQA, label %cQq, label %cQr

cQq:
%lnQB = ptrtoint i64* %Base_Arg to i64
%lnQC = inttopoint i64 %lnQB to i8*
%lnQD = load i64, i64* %l01D
%lnQE = inttopoint i64 %lnQD to i8*
%lnQF = bitcast i8* @newCAF to i8* (i8*, i8*)
store i64 undef, i64* %R3_Var
store i64 undef, i64* %R4_Var
store i64 undef, i64* %R5_Var
store i64 undef, i64* %R6_Var
store float undef, float* %F1_Var
store double undef, double* %D1_Var
store float undef, float* %F2_Var
store double undef, double* %D2_Var
store float undef, float* %F3_Var
store double undef, double* %D3_Var
store float undef, float* %F4_Var
store double undef, double* %D4_Var
store float undef, float* %F5_Var
store double undef, double* %D5_Var
store float undef, float* %F6_Var
store double undef, double* %D6_Var
%lnQC = call ccc i8* (i8*, i8*) %lnQF( i8* %lnQC, i8* %lnQE ) nounwind
%lnQH = ptrtoint i8* %lnQH to i64
store i64 %lnQH, i64* %l1cQm
%lnQI = load i64, i64* %l1cQm
%lnQJ = icmp eq i64 %lnQI, 0
br %i1 %lnQJ, label %cQo, label %cQn

cQn:
%lnQL = ptrtoint i8* @stg_bh_upd_frame_info to i64
 Second Experiment

```plaintext
%lnQK = load i64*, i64** %Sp_Var
%lnQM = getelementptr inbounds i64, i64* %lnQK, i32 -2
store i64 %lnQL, i64* %lnQM, !tbaa !2
%lnQO = load i64, i64* %lcQm
%lnQN = load i64*, i64** %Sp_Var
%lnQP = getelementptr inbounds i64, i64* %lnQN, i32 -1
store i64 %lnQO, i64* %lnQP, !tbaa !2
%lnQQ = ptrtoint %Main_main_closure_struct* @Main_main_closure$def to i64
store i64 %lnQQ, i64* %R2_Var
%lnQR = ptrtoint i8* @base_GHCziTopHandler_runMainIO_closure to i64
store i64 %lnQR, i64* %R1_Var
%lnQS = load i64*, i64** %Sp_Var
%lnQT = getelementptr inbounds i64, i64* %lnQS, i32 -2
%lnQU = ptrtoint i64* %lnQT to i64
%lnQV = inttoptr i64 %lnQU to i64*
store i64* %lnQV, i64** %Sp_Var
%lnQW = inttoptr i64* @stg_ap_p_fast to void (i64*, i64*, i64*, i64, i64, i64,
→ i64, i64, i64, i64)*
%lnQX = load i64*, i64** %Sp_Var
%lnQY = load i64, i64* %R1_Var
%lnQZ = load i64, i64* %R2_Var
tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)
→ %lnQW( i64* %Base_Arg, i64* %lnQX, i64* %Hp_Arg, i64 %lnQY, i64 %lnQZ,
→ i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef ) nounwind
ret void
cQo:
%lnR0 = load i64, i64* %l01D
%lnR1 = inttoptr i64 %lnR0 to i64*
%lnR2 = load i64, i64* %lnR1, !tbaa !1
%lnR3 = inttoptr i64 %lnR2 to void (i64*, i64*, i64*, i64, i64, i64, i64,
→ i64, i64, i64)*
%lnR4 = load i64*, i64** %Sp_Var
%lnR5 = load i64, i64* %R1_Var
tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)
→ %lnR3( i64* %Base_Arg, i64* %lnR4, i64* %Hp_Arg, i64 %lnR5, i64 undef,
→ i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef ) nounwind
ret void
cQq:
%lnR6 = load i64, i64* %l01D
store i64 %lnR6, i64* %l01D
%lnR7 = getelementptr inbounds i64, i64* %Base_Arg, i32 -2
%lnR8 = bitcast i64* %lnR7 to i64*
%lnR9 = load i64, i64* %lnR8, !tbaa !5
%lnRa = inttoptr i64 %lnR9 to void (i64*, i64*, i64*, i64, i64, i64, i64,
→ i64, i64, i64)*
%lnRb = load i64*, i64** %Sp_Var
%lnRc = load i64, i64* %R1_Var
tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)
→ %lnRa( i64* %Base_Arg, i64* %lnRb, i64* %Hp_Arg, i64 %lnRc, i64 undef,
→ i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef, i64 undef ) nounwind
ret void
}
%xP2_bytes_struct = type <([5 x i8]>)
@xP2_bytes$def = internal constant %xP2_bytes_struct<([5 x i8] [i8 109, i8 97,
→ i8 105, i8 110, i8 0])>, align 1
@xP2_bytes = internal alias i8, bitcast (%xP2_bytes_struct* @xP2_bytes$def to i8*)
%xPb_closure_struct = type <(i64, i64)>
@xPb_closure$def = internal global %xPb_closure_struct<(i64 ptrtoint (i8* @ghczmprim_GHCziTypes_TrNameS_con_info to i64), i64 ptrtoint
→ (%xP2_bytes_struct* @xP2_bytes$def to i64)>)
@xPb_closure = internal alias i8, bitcast (%xPb_closure_struct* @xPb_closure$def to i8*)
```

33
The two input programs only have renaming differences, so the second input program is not displayed here.

---

**Generated Files**

Because generated graph, term list and term matching are very large, they will not be shown here.
.2.1. Variable Substitution

| Subst [(Name "lnRc_left", TiVar (Name "lnRc_right")),        |
| (Name "lnRb_left", TiVar (Name "lnRb_right")),          |
| (Name "lnRa_left", TiVar (Name "lnRa_right")),          |
| (Name "lnR9_left", TiVar (Name "lnR9_right")),           |
| (Name "lnR7_left", TiVar (Name "lnR7_right")),           |
| (Name "lnR6_left", TiVar (Name "lnR6_right")),           |
| (Name "lnR5_left", TiVar (Name "lnR5_right")),           |
| (Name "lnR4_left", TiVar (Name "lnR4_right")),           |
| (Name "lnR3_left", TiVar (Name "lnR3_right")),           |
| (Name "lnR2_left", TiVar (Name "lnR2_right")),           |
| (Name "lnR0_left", TiVar (Name "lnR0_right")),           |
| (Name "lnQZ_left", TiVar (Name "lnQZ_right")),           |
| (Name "lnQY_left", TiVar (Name "lnQY_right")),           |
| (Name "lnQX_left", TiVar (Name "lnQX_right")),           |
| (Name "lnQW_left", TiVar (Name "lnQW_right")),           |
| (Name "stg_ap_p_fast_left", TiVar (Name "stg_ap_p_fast_right")), |
| (Name "lnQV_left", TiVar (Name "lnQV_right")),           |
| (Name "lnQU_left", TiVar (Name "lnQU_right")),           |
| (Name "lnQT_left", TiVar (Name "lnQT_right")),           |
| (Name "lnQS_left", TiVar (Name "lnQS_right")),           |
| (Name "lnQR_left", TiVar (Name "lnQR_right")),           |
| (Name "base_GHCziTopHandler_runMainIO_closure_left", TiVar (Name "base_GHCziTopHandler_runMainIO_closure_right")), |
| (Name "lnQQ_left", TiVar (Name "lnQQ_right")),           |
| (Name "Main_main_closure$def_left", TiVar (Name "Main_main_closure$def_right")), |
| (Name "lnQN_left", TiVar (Name "lnQN_right")),           |
| (Name "lnQO_left", TiVar (Name "lnQO_right")),           |
| (Name "lnQK_left", TiVar (Name "lnQK_right")),           |
| (Name "lnQL_left", TiVar (Name "lnQL_right")),           |
| (Name "lnQJ_left", TiVar (Name "lnQJ_right")),           |
| (Name "cQo_left", TiVar (Name "cQo_right")),             |
| (Name "cQn_left", TiVar (Name "cQn_right")),             |
| (Name "lnQI_left", TiVar (Name "lnQI_right")),           |
| (Name "lnQH_left", TiVar (Name "lnQH_right")),           |
| (Name "lnQG_left", TiVar (Name "lnQG_right")),           |
| (Name "D6_Var_left", TiVar (Name "D6_Var_right")),       |
| (Name "F6_Var_left", TiVar (Name "F6_Var_right")),       |
| (Name "D5_Var_left", TiVar (Name "D5_Var_right")),       |
| (Name "F5_Var_left", TiVar (Name "F5_Var_right")),       |
| (Name "D4_Var_left", TiVar (Name "D4_Var_right")),       |
| (Name "F4_Var_left", TiVar (Name "F4_Var_right")),       |
(Name "D3_Var_left", TiVar (Name "D3_Var_right")),
(Name "F3_Var_left", TiVar (Name "F3_Var_right")),
(Name "D2_Var_left", TiVar (Name "D2_Var_right")),
(Name "F2_Var_left", TiVar (Name "F2_Var_right")),
(Name "D1_Var_left", TiVar (Name "D1_Var_right")),
(Name "F1_Var_left", TiVar (Name "F1_Var_right")),
(Name "lnQF_left", TiVar (Name "lnQF_right")),
(Name "newCAF_left", TiVar (Name "newCAF_right")),
(Name "lnQE_left", TiVar (Name "lnQE_right")),
(Name "lnQD_left", TiVar (Name "lnQD_right")),
(Name "lnQC_left", TiVar (Name "lnQC_right")),
(Name "lnQB_left", TiVar (Name "lnQB_right")),
(Name "Base_Arg_left", TiVar (Name "Base_Arg_right")),
(Name "lnQA_left", TiVar (Name "lnQA_right")),
(Name "cQq_left", TiVar (Name "cQq_right")),
(Name "cQr_left", TiVar (Name "cQr_right")),
(Name "SpLim_Arg_left", TiVar (Name "SpLim_Arg_right")),
(Name "lnQz_left", TiVar (Name "lnQz_right")),
(Name "lnQy_left", TiVar (Name "lnQy_right")),
(Name "lnQx_left", TiVar (Name "lnQx_right")),
(Name "lnQw_left", TiVar (Name "lnQw_right")),
(Name "lnQv_left", TiVar (Name "lnQv_right")),
(Name "lnQu_left", TiVar (Name "lnQu_right")),
(Name "cQp_left", TiVar (Name "cQp_right")),
(Name "Sp_Arg_left", TiVar (Name "Sp_Arg_right")),
(Name "R1_Arg_left", TiVar (Name "R1_Arg_right")),
(Name "R6_Var_left", TiVar (Name "R6_Var_right")),
(Name "R5_Var_left", TiVar (Name "R5_Var_right")),
(Name "R4_Var_left", TiVar (Name "R4_Var_right")),
(Name "lnQi_left", TiVar (Name "lnQi_right")),
(Name "lnQh_left", TiVar (Name "lnQh_right")),
(Name "lnQg_left", TiVar (Name "lnQg_right")),
(Name "lnQf_left", TiVar (Name "lnQf_right")),
(Name "lnQd_left", TiVar (Name "lnQd_right")),
(Name "lnQc_left", TiVar (Name "lnQc_right")),
(Name "lnQb_left", TiVar (Name "lnQb_right")),
(Name "lnQa_left", TiVar (Name "lnQa_right")),
(Name "lnQ9_left", TiVar (Name "lnQ9_right")),
(Name "lnQ8_left", TiVar (Name "lnQ8_right")),
(Name "lnQ7_left", TiVar (Name "lnQ7_right")),
(Name "lnQ6_left", TiVar (Name "lnQ6_right")),
(Name "lnQ5_left", TiVar (Name "lnQ5_right")),
(Name "lnQ4_left", TiVar (Name "lnQ4_right")),
(Name "lnQ3_left", TiVar (Name "lnQ3_right")),
(Name "lnQ2_left", TiVar (Name "lnQ2_right")),
(Name "lnQ1_left", TiVar (Name "lnQ1_right")),
(Name "stg_ap_pp_fast_left", TiVar (Name "stg_ap_pp_fast_right")),
(Name "stg_ap_pp_fast_right", TiVar (Name "stg_ap_pp_fast_left")),
.2.2. Line Matching and Report
ppMatchedprogram_hide

0 target datalayout = "e-m:e-i64:6
1 target triple = "x86_64-unknown-
2 declare ccc i8* @memcpysdef(i8
3 declare ccc i8* @memmove$def
4 declare ccc i8* @memset$def(i8
5 declare ccc i64 @newSpark$def
6 !0 = !["root"
7 !1 = !["top", !0]
8 !2 = !["stack", !1]
9 !3 = !["heap", !1]
10 !4 = !["rx", !3]
11 !5 = !["base", !1]
12
13 %Spk_closure_struct = type <{i6
14 @Spk_closure$def = internal glc
15 @Spk_closure = internal alias i8
16 %, uPu_srt_struct = type <{i64, i
17 %Main_main_closure struct = ty
18 @ _uPu_srt$def = internal global
19 @ _uPu_srt = internal alias i8, b
20 @Main_main_closure$def = inte
21 @Main_main_closure = alias i8,
22 @Main_main_info = alias i8, bitc
define ghccc void @Main_main_
24 {
25 nPv:
26 %lr1 = alloca i64, i32 1
27 %R3_Var = alloca i64, i32 1
28 store i64 undef, i64* %R3_Var
29 %R4_Var = alloca i64, i32 1
30 store i64 undef, i64* %R4_Var
31 %R5_Var = alloca i64, i32 1
32 store i64 undef, i64* %R5_Var
33 %R6_Var = alloca i64, i32 1
34 store i64 undef, i64* %R6_Var
35 %F1_Var = alloca float, i32 1
36 store float undef, float* %F1_Vt
37 %D1_Var = alloca double, i32 1
38 store double undef, double* %
39 %F2_Var = alloca i32 1
40 store float undef, float* %F2_Vt
41 %D2_Var = alloca double, i32 1
42 store double undef, double* %
43 %F3_Var = alloca float, i32 1
44 store float undef, float* %F3_Vt
45 %D3_Var = alloca double, i32 1
46 store double undef, double* %
47 %F4_Var = alloca float, i32 1
48 store float undef, float* %F4_Vt

Second Experiment
Second Experiment
ppMatchedprogram_hide

98  store double undef, double* %l
99  %lnP = call ccc i8* (i8*, i8*)
100  %lnPK = ptrtoint i8* %lnP to i
101  store i64 %lnPK, i64* %lcPo
102  %lnPL = load i64, i64* %lcPo
103  %lnPM =icmp eq i64 %lnPL, 0
104  br i1 %lnPM, label %cPq, label
105  cPp:
106  %lnPO = ptrtoint i8* @stg_bh_
107  %lnPN = load i64*, i64* %Sp_
108  %lnPP = getelementptr inbound
109  store i64 %lnPO, i64* %lnPP, !t
110  %lnPR = load i64, i64* %lnPO
111  %lnPQ = load i64*, i64* %Sn
112  %lnPS = getelementptr inbound
113  store i64 %lnPR, i64* %lnPS, !t
114  %lnPT = ptrtoint %Spk_closure
115  %lnPU = add i64 %lnPT, 1
116  store i64 %lnPU, i64* %R3_Var
117  %lnPV = ptrtoint i8* @base_G
118  store i64 %lnPV, i64* %R2_Var
119  %lnPW = ptrtoint i8* @base_S
120  store i64 %lnPW, i64* %R1_Va
121  %lnPX = load i64*, i64* %Sp_
122  %lnPY = getelementptr inbound
123  %lnPZ = ptrtoint i64* %lnPY
to
124  %lnQ0 = inttoptr i64 %lnQ0 to i
125  store i64* %lnQ0, i64** %Sp_V
126  %lnQ1 = bitcast i8* @stg_ap_p
127  %lnQ2 = load i64*, i64** %Sp_ to
128  %lnQ3 = load i64, i64* %R1_Vi
129  %lnQ4 = load i64, i64* %R2_Vi
130  %lnQ5 = load i64, i64* %R3_Vi
tail call ghcvoid void (i64*, i64
tail call ghcvoid void (i64*, i64
132  ret void
133  cPp:
134  %lnQ6 = load i64, i64* %lr
cPp:
135  %lnQ7 = inttoptr i64 %lnQ6 to i
136  %lnQ8 = load i64, i64* %lnQ7,
137  %lnQ9 = inttoptr i64 %lnQ8 to
138  %lnQa = load i64*, i64* %lnQ9
tail call ghcvoid void (i64*, i64
tail call ghcvoid void (i64*, i64
141  ret void
142  cPp:
143  %lnQc = load i64, i64* %lr
cPp:
144  store i64 %lnQc, i64* %R1_Va
145  %lnQd = getelementptr inbound
%lnQe = bitcast i64* %lnQd to
%lnQe = bitcast i64* %lnQd to

Second Experiment
Second Experiment

\[ \text{ppMatchedProgram} \_\text{hide} \]

\[ \text{define cc1 @llvm.expect.i1(i1, i32)} \]

\[ \text{@ZCMain_main_info = alias i8, bit} \]

\[ \text{@ZCMain_main_closure = alias i} \]

\[ \text{@ZCMain_main_closure$def = intern} \]

\[ \text{@_uQs_srt = internal alias i8, b} \]

\[ \text{@_uQs_srt$def = internal global} \]

\[ \text{%ZCMain_main_closure_struct = ty} \]

\[ \text{%_uQs_srt_struct = type <{i64, i64}>} \]

\[ \text{declare ccc i1 @llvm.expect.i1(i1, i32)} \]

\[ \text{ret void} \]

\[ \text{tail call ghccc void (i64*, i64* %lnQi)} \]

\[ \text{%lnQi = load i64, i64* %R1_Va} \]

\[ \text{%lnQh = load i64*, i64** %Sp_} \]

\[ \text{%lnQf = load i64, i64* %lnQe, !} \]

\[ \text{store float undef, float* %F6_Va} \]

\[ \text{%F6_Var = alloca float, i32 1} \]

\[ \text{store double undef, double* %D5_Var} \]

\[ \text{%D5_Var = alloca double, i32 1} \]

\[ \text{store float undef, float* %F5_Va} \]

\[ \text{%F5_Var = alloca float, i32 1} \]

\[ \text{store double undef, double* %D4_Var} \]

\[ \text{%D4_Var = alloca double, i32 1} \]

\[ \text{store float undef, float* %F4_Va} \]

\[ \text{%F4_Var = alloca float, i32 1} \]

\[ \text{store double undef, double* %D3_Var} \]

\[ \text{%D3_Var = alloca double, i32 1} \]

\[ \text{store float undef, float* %F3_Va} \]

\[ \text{%F3_Var = alloca float, i32 1} \]

\[ \text{store double undef, double* %D2_Var} \]

\[ \text{%D2_Var = alloca double, i32 1} \]

\[ \text{store float undef, float* %F2_Va} \]

\[ \text{%F2_Var = alloca float, i32 1} \]

\[ \text{store double undef, double* %D1_Var} \]

\[ \text{%D1_Var = alloca double, i32 1} \]

\[ \text{store float undef, float* %F1_Va} \]

\[ \text{%F1_Var = alloca float, i32 1} \]

\[ \text{store i64 undef, i64* %R6_Var} \]

\[ \text{%R6_Var = alloca i64, i32 1} \]

\[ \text{store i64 undef, i64* %R5_Var} \]

\[ \text{%R5_Var = alloca i64, i32 1} \]

\[ \text{store i64 undef, i64* %R4_Var} \]

\[ \text{%R4_Var = alloca i64, i32 1} \]

\[ \text{store i64 undef, i64* %R3_Var} \]

\[ \text{%R3_Var = alloca i64, i32 1} \]

\[ \text{%l01D = alloca i64, i32 1} \]

\[ \text{nQt:} \]

\[ \text{define ghccc void @ZCMain_main_i} \]

\[ \text{@ZCMain_main_info = alias i8, bit} \]

\[ \text{@ZCMain_main_closure = alias i} \]

\[ \text{@ZCMain_main_closure$def = intern} \]

\[ \text{@_uQs_srt = internal alias i8, b} \]

\[ \text{@_uQs_srt$def = internal global} \]

\[ \text{%ZCMain_main_closure_struct = ty} \]

\[ \text{%_uQs_srt_struct = type <{i64, i64}>} \]

\[ \text{declare ccc i1 @llvm.expect.i1(i1, i32)} \]

\[ \text{ret void} \]

\[ \text{tail call ghccc void (i64*, i64* %lnQi)} \]

\[ \text{%lnQi = load i64, i64* %R1_Va} \]

\[ \text{%lnQh = load i64*, i64** %Sp_} \]
ppMatchedprogram_hide

196 %D6_Var = alloca double, i32 1  %D6_Var = alloca double, i32 1
197 store double undef, double* %D6_Var  store double undef, double* %D6_Var = alloca double, i32 1
198 store double undef, double* %D6_Var  store double undef, double* %D6_Var
199 %R2_Var = alloca i64, i32 1  %R2_Var = alloca i64, i32 1
200 store i64 undef, i64* %R2_Var  store i64 undef, i64* %R2_Var
201 %R1_Var = alloca i64, i32 1  %R1_Var = alloca i64, i32 1
202 store i64 %R1_Arg, i64* %R1_Var  store i64 %R1_Arg, i64* %R1_Var
203 %Sp_Var = alloca i64*, i32 1  %Sp_Var = alloca i64*, i32 1
204 store i64* %Sp_Arg, i64** %Sp  store i64* %Sp_Arg, i64** %Sp
205 br label %cQP  br label %cQP
206 cQP:
207 %lnQu = load i64, i64* %lnQu  %lnQu = load i64, i64* %lnQu
208 store i64 %lnQu, i64* %lnQu  store i64 %lnQu, i64* %lnQu
209 %lnQv = load i64, i64* %lnQv  %lnQv = load i64, i64* %lnQv
210 %lnQw = getelementptr inbounds %lnQw to %lnQw = getelementptr inbounds %lnQw to
211 %lnQz = icmp ult i64 %lnQz, 24 %lnQz = icmp ult i64 %lnQz, 24
212 %lnQC = inttoptr i64 %lnQC to %lnQC = inttoptr i64 %lnQC to
213 %lnQA = call ccc i1 (i1, i1) @ll %lnQA = call ccc i1 (i1, i1) @ll
214 %lnQB = ptrtoint i64* %lnQB to %lnQB = ptrtoint i64* %lnQB to
215 br i1 %lnQA, label %cQP, label %cQP  br i1 %lnQA, label %cQP, label %cQP
216 cQP:
217 %lnQB = ptrtoint i64* %lnQB to %lnQB = ptrtoint i64* %lnQB to
218 %lnQC = inttoptr i64 %lnQC to %lnQC = inttoptr i64 %lnQC to
219 %lnQD = load i64, i64* %lnQD to %lnQD = load i64, i64* %lnQD to
220 %lnQE = inttoptr i64 %lnQE to %lnQE = inttoptr i64 %lnQE to
221 %lnQF = bitcast i8* @newCAF to %lnQF = bitcast i8* @newCAF
222 store i64 undef, i64* %R3_Var  store i64 undef, i64* %R3_Var
223 store i64 undef, i64* %R4_Var  store i64 undef, i64* %R4_Var
224 store i64 undef, i64* %R5_Var  store i64 undef, i64* %R5_Var
225 store i64 undef, i64* %R6_Var  store i64 undef, i64* %R6_Var
226 store float undef, float* %F1_Vi  store float undef, float* %F1_Vi
227 store double undef, double* %F1_Vi  store double undef, double* %F1_Vi
228 store float undef, float* %F2_Vi  store float undef, float* %F2_Vi
229 store double undef, double* %F2_Vi  store double undef, double* %F2_Vi
230 store float undef, float* %F3_Vi  store float undef, float* %F3_Vi
231 store double undef, double* %F3_Vi  store double undef, double* %F3_Vi
232 store float undef, float* %F4_Vi  store float undef, float* %F4_Vi
233 store double undef, double* %F4_Vi  store double undef, double* %F4_Vi
234 store float undef, float* %F5_Vi  store float undef, float* %F5_Vi
235 store double undef, double* %F5_Vi  store double undef, double* %F5_Vi
236 store float undef, float* %F6_Vi  store float undef, float* %F6_Vi
237 store double undef, double* %F6_Vi  store double undef, double* %F6_Vi
238 %lnQG = call ccc i8* (i8*, i8*) %lnQG = call ccc i8* (i8*, i8*)
239 %lnQH = ptrtoint i8* %lnQG to %lnQH = ptrtoint i8* %lnQG to
240 store i64 %lnQH, i64* %lnQH  store i64 %lnQH, i64* %lnQH
241 %lnQI = load i64, i64* %lnQI  %lnQI = load i64, i64* %lnQI
242 %lnQJ = icmp eq i64 %lnQJ  %lnQJ = icmp eq i64 %lnQJ
243 br i1 %lnQJ, label %cQo, label %cQo  br i1 %lnQJ, label %cQo, label %cQo
244 cQn:
Second Experiment

ppMatchedprogram_hide

```assembly
245 %lnQL = ptrtoint i8* @stg_bh_u  %lnQL = ptrtoint i8* @ 245 ###   %lnQL = ptrtoint ... ### %rPb_closure_struct = type <{i64           
293 @rPb_closure$def = internal glo  ### @rPb_closure$def = internal glo
```

---

Page 6
ppMatchedprogram_hide

294 @rPb_closure = internal alias i8
295 %rPc_bytes_struct = type <{[5 x
296 @rPc_bytes$def = internal consi
297 @rPc_bytes = internal alias i8,
298 %rPd_closure_struct = type <{[6-
299 @rPd_closure$def = internal glo
300 @rPd_closure = internal alias i8
301 %Main_zdtrModule_closure_str.
302 @Main_zdtrModule_closure$del
303 %Main_zdtrModule_closure = al
304 @integerzmwiredzmin_GHCziln
305 @stg_SRT_3_info = external glc
306 @base_SystemziO_print_closu
307 @base_GHCziShow_zdfShowLin
308 @newCAF = external global i8
309 @stg_bh_upd_frame_info = exte
310 @stg_ap_pp_fast = external glol
311 @stg_SRT_2_info = external glc
312 @base_GHCziTopHandler_runM
313 @stg_ap_p_fast = external globi
314 @ghczmprim_GHCziTypes_TrN
315 @ghczmprim_GHCziTypes_Moc
316 @llvm.used = appending consta

### @rPb_closure = internal alias i8
### %rPc_bytes_struct = type <{[5 x
### @rPc_bytes$def = internal consi
### @rPc_bytes = internal alias i8,
### %rPd_closure_struct = type <{[6-
### @rPd_closure$def = internal glo
### @rPd_closure = internal alias i8
### %Main_zdtrModule_closure_str.
### @Main_zdtrModule_closure$del
### %Main_zdtrModule_closure = al
### @integerzmwiredzmin_GHCziln
### @stg_SRT_3_info = external glc
### @base_SystemziO_print_closu
### @base_GHCziShow_zdfShowLin
### @newCAF = external global i8
### @stg_bh_upd_frame_info = exte
### @stg_ap_pp_fast = external glol
### @stg_SRT_2_info = external glc
### @base_GHCziTopHandler_runM
### @stg_ap_p_fast = external globi
### @ghczmprim_GHCziTypes_TrN
### @ghczmprim_GHCziTypes_Moc
### @llvm.used = appending consta

SECOND EXPERIMENT