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
https://digitalcommons.dartmouth.edu/masters_theses/45

This Thesis (Master's) is brought to you for free and open access by the Theses, Dissertations, and Graduate Essays at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Master's Theses and Essays 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

Exchanging 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References</td>
<td>26</td>
</tr>
<tr>
<td>Second Experiment</td>
<td>28</td>
</tr>
<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>
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.
Chapter 2

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

lines from two programs.

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
Finding a Matching Method and Implementation

```haskell
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
3.1 Finding a Matching Method and Implementation

--- source code
store i32 0, i32* %y1, align 4

--- Haskell AST:
Do (Store {volatile = False, address = LocalReference (PointerType
        {pointerReferent = IntegerType {typeBits = 32}, pointerAddrSpace =
        AddrSpace 0}) (UnName y1), value = ConstantOperand (Int {integerBits =
        32, integerValue = 0}), maybeAtomicity = Nothing, alignment = 4,
        metadata = []})

--- converted command
TiApp 0 (Other (Name "Store")) [TiApp 1 (Arguments 2) [Var (Name "y1"),
            Const (ConstantOperand (Int {integerBits = 32, integerValue = 0})))
            Just (Ins (Do (Store {volatile = False, address = LocalReference
            (PointerType {pointerReferent = IntegerType {typeBits = 32},
            pointerAddrSpace = AddrSpace 0}) (UnName y1), value = ConstantOperand
            (Int {integerBits = 32, integerValue = 0}), maybeAtomicity =
            Nothing, alignment = 4, metadata = [])))]

Listing 2: Example of Term

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

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 \texttt{Maybe OriginalAST} is designed for. In the beginning, our idea was to write a transformation function that transforms \texttt{TermIndex} back to Haskell AST datatype. Soon we found that when converting AST to \texttt{TermIndex}, there was lots of information lost because we do not need them for comparison. Therefore, we decide to add a field called \texttt{OriginalAST} to the \texttt{TermIndex} to hold the Haskell AST value of a term. When comparing the equivalence of two terms, \texttt{OriginalAST} is not considered. The reason \texttt{OriginalAST} is a \texttt{Maybe} type is that some terms like block number do not need to store their original AST, so the value of those terms’ \texttt{OriginalAST} field is \texttt{Nothing}.

Having this \texttt{OriginalAST}, we can translate \texttt{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 \texttt{TermIndex} and extract their \texttt{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.
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

```
--program1 --program2
x1 = 1  x2 = 1
y1 = 2  y2 = 2
f(x1)  x2 = 3
x1 = 3  f(x2)

-- constraints table of program2
  defDef  defUse  Escape
  x2=1    x2=3    f(x2)
y2=2
x2=3    f(x2)

--variable matching
[(x1,x2),(y1,y2)]

-- line matching
[(1,1),(2,2),(3,4),(4,3)]
```

Listing 3: Example of Invalid Matching

rules to check if some matched part violates the constraints. Just like the example in Listing 3, the matched lines of second program are [1,2,4,3]. Based on the constraints table, line 3 should occur before line 4; thus, this matching is invalid. If everything satisfies, then we calculate the score and output the report. Otherwise, we generate a warning message to tell the user that there might be errors occur and told them which part of matching causes this error. Users can decide to keep using this substitution and generate the report or adjust the substitution manually. For instance, in Listing 4, the generated variable matching is [(x1,y2),(y1,x2),(z1,z2)], expect user may think x1 should be substituted with x2. Thus, in the generated substitution file, the user can change this matching manually to [(x1,x2),(y1,y2),(z1,z3)]. The line matching after changing substitution would become [(1,1),(2,2),(3,3)].

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

```plaintext
--program1   --program2
x1 = 2       y2 = 2
y1 = 2       x2 = 2
z1 = 3       z2 = 3
```

--matching

```plaintext
[(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 \texttt{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 \( \text{scores} = \text{map(score(terms1, terms2, subst)))matching} \)

From the definition, \texttt{calculateScore} takes four arguments: two term lists, a term matching and a substitution set. It passes these four arguments to a function \texttt{score}. The definition of \texttt{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 \texttt{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 \texttt{calculateScore}, we get the sum of the score list and the product of the score list. The sum is the maximum score \texttt{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 renaming 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 11 has a matched line 12 in matching, then find the index of 12 in the second position. The subtraction of the index of 11 and the index of 12 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
3.2 Generating an Output Method and Implementation

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.

![Figure 3.2: Example of Reordering Program](image)

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...
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.
Chapter 4

Experiments

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
Chapter 5

Discussion

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.

![Figure 5.1: Runtime of Experiments](image)

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.
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


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

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

@Main_main_closure$def = internal global %Main_main_closure_struct<{i64 ptrtoint (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @Main_main_closure$def to i64), i64 0, i64 0, i64 0}>
@Main_main_closure = alias i8, bitcast (%Main_main_closure_struct* @Main_main_closure$def to i8*)
@Main_main_info = alias i8, bitcast (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @Main_main_info$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
  
  cPr:
    %lnPw = load i64, i64* %R1_Var
    store i64 %lnPw, i64* %lrgc
    %lnPx = load i64*, i64** %Sp_Var
    %lnPy = getelementptr inbounds i64, i64* %lnPx, i32 1
    %lnPz = ptrtoint i64* %lnPy to i64
    %lnPA = sub i64 %lnPz, 24
    %lnPB = icmp ult i64 %lnPA, %SpLim_Arg
    %lnPD = call ccc i1 (i1, i1) @llvm.expect.i1( i1 %lnPB, i1 0 )
    br i1 %lnPD, label %cPs, label %cPt
  
  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*)
st ore 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 float undef, float* %F2_Var
store double undef, double* %D3_Var
store float undef, float* %F3_Var
store double undef, double* %D4_Var
store float undef, float* %F4_Var
store double undef, double* %D5_Var
store float undef, float* %F5_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

%lnP0 = 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 %lnP0, i64* %lnPP, !tbaa !2
%lnPR = load i64, i64* %lcPo
%lnPS = load i64*, i64** %Sp_Var
%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
%lnPW = ptrtoint i8* @base_SystemziIO_print_closure to i64
store i64 %lnPW, 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 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* %R1_Var
%lnQ3 = load i64, i64* %R2_Var
%lnQ4 = load i64, i64* %R3_Var
%lnQ5 = load i64, i64* %R3_Var
tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64, i64, i64)
                               → %lnQ1( i64* %Base_Arg, i64* %lnQ2, i64* %Hp_Arg, i64 %lnQ3, i64 %lnQ4,
                               → i64 %lnQ5, i64 undef, i64 undef, i64 undef, i64 %SpLim_Arg ) nounwind
ret void
cPq:
%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
%lnQb = load i64, i64* %R1_Var  
tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)  
→ %lnQ9( i64* %Base_Arg, i64* %lnQa, i64* %Hp_Arg, i64 %lnQb, i64 undef,  
→ i64 undef, i64 undef, i64 undef, i64 undef, i64 %SpLim_Arg ) nounwind  
ret void  
cPd:  
%lnQc = load i64, i64* %lrgc  
store i64 %lnQc, i64* %R1_Var  
%lnQd = getelementptr inbounds i64, i64* %Base_Arg, i32 -2  
%lnQe = bitcast i64* %lnQd to i64*  
%lnQf = load i64, i64* %lnQe, !tbaa !5  
%lnQg = inttoptr i64 %lnQf to void (i64*, i64*, i64*, i64, i64, i64, i64,  
→ i64, i64, i64)*  
%lnQh = load i64*, i64** %Sp_Var  
%lnQi = load i64, i64* %R1_Var  
tail call ghccc void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)  
→ %lnQ9( i64* %Base_Arg, i64* %lnQh, i64* %Hp_Arg, i64 %lnQi, i64 undef,  
→ i64 undef, i64 undef, i64 undef, i64 undef, i64 %SpLim_Arg ) nounwind  
ret void  
}}  
declare ccc i1 @llvm.expect.i1(i1, i1)  
%uQs_srt_struct = type <i64, i64, i64, i64>  
%ZCMain_main_closure_struct = type <i64, i64, i64, i64>  
@uQs_srt$def = internal global %uQs_srt_struct<i64 ptrtoint (i8* @stg_SRT_2_info to i64), i64 ptrtoint (i8* @base_GHCziTopHandler_runMainIO_closure to i64), i64 ptrtoint to i64>, i64 0>  
@uQs_srt = internal alias i8, bitcast (%uQs_srt_struct* @uQs_srt$def to i8*)  
@ZCMain_main_closure$def = internal global %ZCMain_main_closure_struct<i64 ptrtoint (void (i64*, i64*, i64*, i64, i64, i64, i64, i64, i64, i64)* @ZCMain_main_info$def to i64), i64 0, i64 0, i64 0>  
@ZCMain_main_closure = alias i8, bitcast (%ZCMain_main_closure_struct* @ZCMain_main_closure$def to i8*)  
@ZCMain_main_info = alias i8, bitcast (%ZCMain_main_closure_struct* @ZCMain_main_closure to i8*)  
define ghccc void @ZCMain_main_info$def(i64* noalias nocapture %Base_Arg, i64* noalias nocapture %Sp_Arg, i64* noalias nocapture %Hp_Arg, i64 %R1_Arg,  
→ i64 %R2_Arg, i64 %R3_Arg, i64 %R4_Arg, i64 %R5_Arg, i64 %R6_Arg, i64 %SpLim_Arg) align 8 nounwind prefix <(i64 132, 132)<<(i64 0, 132 21, i32 44 132 trunc (i64 sub (i64 ptrtoint (%uQs_srt_struct* @uQs_srt$def to i64),i64 0) ptrtoint to i64),i64 0)>) to i32), i32 0>  
{  
Q7:  
%l01D = 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  
}  
convert nocapture %Base_Arg to i1  
%l10D = alloca i64, i32 1  
%R1_Var = alloca i64, i32 1  
store i64 undef, i64* %R1_Var
Second Experiment

```
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

%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

%lnQG = call ccc i8* (i8*, i8*) %lnQF( i8* %lnQC, i8* %lnQD ) nounwind
%lnQH = ptrtoint i8* %lnQG to i64
store i64 %lnQH, i64* %lcQm
%lnQI = load i64, i64* %lcQm
%lnQJ = icmp eq i64 %lnQI, 0
br i1 %lnQJ, label %cQo, label %cQn

%lnQL = ptrtoint i8* @stg_bh_upd_frame_info to i64
```

32
Second Experiment

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

---

Section 2

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 "lnQ0_left",TiVar (Name "lnQ0_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")),

35
SECOND EXPERIMENT
SECOND EXPERIMENT

(Name "lnQ0_left",TiVar (Name "lnQ0_right")),
(Name "lnPZ_left",TiVar (Name "lnPZ_right")),
(Name "lnPY_left",TiVar (Name "lnPY_right")),
(Name "lnPX_left",TiVar (Name "lnPX_right")),
(Name "lnPW_left",TiVar (Name "lnPW_right")),
(Name "base_SystemziIO_print_closure_left",TiVar (Name "base_SystemziIO_print_closure_right")),
(Name "lnPV_left",TiVar (Name "lnPV_right")),
(Name "base_GHCziShow_zdfShowInteger_closure_left",TiVar (Name "base_GHCziShow_zdfShowInteger_closure_right")),
(Name "R3_Var_left",TiVar (Name "R3_Var_right")),
(Name "lnPU_left",TiVar (Name "lnPU_right")),
(Name "lnPT_left",TiVar (Name "lnPT_right")),
(Name "sPk_closure$def_left",TiVar (Name "sPk_closure$def_right")),
(Name "lnPQ_left",TiVar (Name "lnPQ_right")),
(Name "lnPR_left",TiVar (Name "lnPR_right")),
(Name "lnPN_left",TiVar (Name "lnPN_right")),
(Name "lnPO_left",TiVar (Name "lnPO_right")),
(Name "lnPM_left",TiVar (Name "lnPM_right")),
(Name "cPq_left",TiVar (Name "cPq_right")),
(Name "cPp_left",TiVar (Name "cPp_right")),
(Name "lnPL_left",TiVar (Name "lnPL_right")),
(Name "lnPK_left",TiVar (Name "lnPK_right")),
(Name "lnPI_left",TiVar (Name "lnPI_right")),
(Name "lnPH_left",TiVar (Name "lnPH_right")),
(Name "lnPG_left",TiVar (Name "lnPG_right")),
(Name "lnPF_left",TiVar (Name "lnPF_right")),
(Name "lnPE_left",TiVar (Name "lnPE_right")),
(Name "lnPD_left",TiVar (Name "lnPD_right")),
(Name "cPs_left",TiVar (Name "cPs_right")),
(Name "cPt_left",TiVar (Name "cPt_right")),
(Name "lnPB_left",TiVar (Name "lnPB_right")),
(Name "lnPA_left",TiVar (Name "lnPA_right")),
(Name "lnPz_left",TiVar (Name "lnPz_right")),
(Name "lnPy_left",TiVar (Name "lnPy_right")),
(Name "lnPx_left",TiVar (Name "lnPx_right")),
(Name "lnPw_left",TiVar (Name "lnPw_right")),
(Name "cPr_left",TiVar (Name "cPr_right")),
(Name "lnPJ_left",TiVar (Name "lnPJ_right")),
(Name "stg_bh_upd_frame_info_left",TiVar (Name "stg_bh_upd_frame_info_right")),
(Name "lnQM_left",TiVar (Name "lnQM_right")),
(Name "lnQP_left",TiVar (Name "lnQP_right")),
(Name "lnPS_left",TiVar (Name "lnPS_right")),
(Name "lnQ7_left",TiVar (Name "lnQ7_right")),
(Name "lnQe_left",TiVar (Name "lnQe_right")),

37
### 2.2. Line Matching and Report

```
[(26,26), (166,166), (222,222), (168,168), (223,223), (170,170), (224,224), (172,172),
(225,225), (174,174), (226,226), (176,176), (227,227), (178,178), (228,228), (180,180),
(229,229), (182,182), (230,230), (184,184), (231,231), (186,186), (232,232), (188,188),
(233,233), (190,190), (234,234), (192,192), (235,235), (194,194), (236,236), (196,196),
(59,59), (199,199), (200,200), (201,201), (202,202), (203,203), (204,204), (66,66),
(68,68), (69,69), (70,70), (72,72), (73,73), (74,74), (76,76), (78,78), (79,79), (80,80),
(81,81), (82,82), (167,167), (169,169), (171,171), (173,173), (175,175), (177,177),
(179,179), (181,181), (183,183), (185,185), (187,187), (189,189), (191,191), (193,193),
(195,195), (197,197), (200,200), (201,201), (202,202), (203,203), (204,204), (267,267),
(268,268), (269,269), (270,270), (271,271), (272,272), (273,273), (274,274), (275,275),
(276,276), (277,277), (278,278), (279,279), (280,280), (281,281), (282,282), (283,283),
(284,284), (285,285), (132,132)]
```
0 target datalayout = "e-m:e-i64:6
1 target triple = "x86_64-unknown-
2 declare ccc i8* @memc$p$def(ii
3 declare ccc i8* @memmove$def
4 declare ccc i8* @memset$def(ii
5 declare ccc i64 @newSpark$def
6 l! = ![""root"]
7 l! = ![""top", ![0]
8 l! = ![""stack", ![1]
9 l! = ![""heap", ![1]
10 l! = !["rx", ![3]
11 l! = !["base", ![1]
12
13 %sPk_closure_struct = type <i6
14 @sPk_closure$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
23 define ghccc void @Main_main_
24 {
25 nPv:
26 %lr1 = alloca i64, i32 1 %lr1 = alloca i64, i32 1
27 %R3_Var = alloca i64, i32 1 %R3_Var = alloca i64, i32 1
28 store i64 undef, i64* %R3_Var store i64 undef, i64*
29 %R4_Var = alloca i64, i32 1 %R4_Var = alloca i64, i64*
30 store i64 undef, i64* %R4_Var store i64 undef, i64*
31 %R5_Var = alloca i64, i32 1 %R5_Var = alloca i64, i32 1
32 store i64 undef, i64* %R5_Var store i64 undef, i64*
33 %R6_Var = alloca i64, i32 1 %R6_Var = alloca i64, i64*
34 store i64 undef, i64* %R6_Var store i64 undef, i64*
35 %F1_Var = alloca float, i32 1 %F1_Var = alloca float
36 store float undef, float* %F1_V
37 %D1_Var = alloca double, i32 1 %D1_Var = alloca doul
38 store double undef, double* % store double undef, double* %
39 %F2_Var = alloca float, i32 1 %F2_Var = alloca float,
40 store float undef, float* %F2_V store float undef, float
41 %D2_Var = alloca double, i32 1 %D2_Var = alloca double
42 store double undef, double* % store double undef, double* %
43 %F3_Var = alloca float, i32 1 %F3_Var = alloca float,
44 store float undef, float* %F3_V store float undef, float*
45 %D3_Var = alloca double, i32 1 %D3_Var = alloca double
46 store double undef, double* % store double undef, double* %
47 %F4_Var = alloca float, i32 1 %F4_Var = alloca float,
48 store float undef, float* %F4_V store float undef, float* %F4_V

Second Experiment
ppMatchedprogram_hide

%D4_Var = alloca double, i32 1
store double undef, double* %D4_Var = alloca double, i32 1
store double undef, double* %D4_Var = alloca double, i32 1
50  store double undef, d 50  store double undef, double* %D4_Var = alloca double, i32 1
51  %F5_Var = alloca float, i32 1 51  %F5_Var = alloca float, i32 1
52  store float undef, float* %F5_Vi 52  store float undef, float* %F5_Vi 53  %D5_Var = alloca double, i32 1 53  %D5_Var = alloca double, i32 1
54  store double undef, double* %D5_Var = alloca double, i32 1 54  store double undef, double* %D5_Var = alloca double, i32 1
55  %F6_Var = alloca float, i32 1 55  %F6_Var = alloca float, i32 1
56  store float undef, float* %F6_Vi 56  store float undef, float* %F6_Vi 57  %D6_Var = alloca double, i32 1 57  %D6_Var = alloca double, i32 1
58  store double undef, double* %D6_Var = alloca double, i32 1 58  store double undef, double* %D6_Var = alloca double, i32 1
59  %scPo = alloca i64, i32 1 59  %scPo = alloca i64, i32 1
60  %R2_Var = alloca i64, i32 1 60  %R2_Var = alloca i64, i32 1
61  store i64 undef, i64* %R2_Var 61  store i64 undef, i64* %R2_Var
62  %R1_Var = alloca i64, i32 1 62  %R1_Var = alloca i64, i32 1
63  store i64 undef, i64* %R1_Var 63  store i64 undef, i64* %R1_Var
64  %Sp_Var = alloca i64*, i32 1 64  %Sp_Var = alloca i64*, i32 1
65  store i64* %Sp_Arg, i64** %Sp 65  store i64* %Sp_Arg, i64** %Sp
66  br label %cPr 66  br label %cPr
67  cP:
68  %lnPw = load i64, i64* %R1_Vi 68  %lnPw = load i64, i64* %R1_Vi
69  store i64 %lnPw, i64* %lnPw 69  store i64 %lnPw, i64* %lnPw
70  %lnPx = load i64*, i64** %Sp \ 70  %lnPx = load i64*, i64** %Sp \ 71  %lnPy = getelementptr inbounds 71  %lnPy = getelementptr inbounds
72  %lnPz = pptrtoint i64* %lnPy to i 72  %lnPz = pptrtoint i64* %lnPy to i
73  %lnPA = sub i64 %lnPz, 24 73  %lnPA = sub i64 %lnPz, 24
74  %lnPB = icmp ult i64 %lnPA 74  %lnPB = icmp ult i64 %lnPA
75  %lnPD = call ccc i1 (i1, i1) @ 75  %lnPD = call ccc i1 (i1, i1) @
76  br i1 %lnPD, label %cPs, label 76  br i1 %lnPD, label %cPs, label
77  cP:
78  %lnPE = pptrtoint i64* %Base_A 78  %lnPE = pptrtoint i64* %Base_A
79  %lnPF = inttoptr i64 %lnPE to i 79  %lnPF = inttoptr i64 %lnPE to i
80  %lnPG = load i64, i64* %lnPF 80  %lnPG = load i64, i64* %lnPF
81  %lnPH = inttoptr i64 %lnPG to i 81  %lnPH = inttoptr i64 %lnPG to i
82  %lnPI = bitcast i8* @newCAF t 82  %lnPI = bitcast i8* @newCAF t
83  store i64 undef, i64* %R3_Var 83  store i64 undef, i64* %R3_Var
84  store i64 undef, i64* %R4_Var 84  store i64 undef, i64* %R4_Var
85  store i64 undef, i64* %R5_Var 85  store i64 undef, i64* %R5_Var
86  store i64 undef, i64* %R6_Var 86  store i64 undef, i64* %R6_Var
87  store float undef, float* %F1_Vi 87  store float undef, float* %F1_Vi
88  store double undef, double% 88  store double undef, double% 
89  store float undef, float* %F2_Vi 89  store float undef, float* %F2_Vi
90  store double undef, double% 90  store double undef, double% 
91  store float undef, float* %F3_Vi 91  store float undef, float* %F3_Vi
92  store double undef, double% 92  store double undef, double% 
93  store float undef, float* %F4_Vi 93  store float undef, float* %F4_Vi
94  store double undef, double% 94  store double undef, double% 
95  store float undef, float* %F5_Vi 95  store float undef, float* %F5_Vi
96  store double undef, double% 96  store double undef, double% 
97  store float undef, float* %F6_Vi 97  store float undef, float* %F6_Vi
98  br label %cPr 98  br label %cPr
ppMatchedprogram_hide

98 store double undef, double* % 98 store double undef, d 98 98 store double undef, double* %
99 %lnPJ = call ccc i8* (i8*, i8*) 99 %lnPJ = call ccc i8* (i8*, i8*)
100 %lnPK = ptrtoint i8* %lnPJ to i %lnPK = ptrtoint i8* %lnPJ to i
101 store i64 %lnPK, i64* %lnCPo store i64 %lnPK, i64* %lnCPo
102 %lnPL = load i64, i64* %lnCPo %lnPL = load i64, i64* %lnCPo
103 %lnPM = icmp eq i64 %lnPL, 0 %lnPM = icmp eq i64 %lnPL, 0
104 br i1 %lnPM, label %cPq, label br i1 %lnPM, label %cPq, label
105 cPp:
106 %lnPO = ptrtoint i8* @stg_bh_ %lnPO = ptrtoint i8* @stg_bh_
107 %lnPN = load i64*, i64** %cPp %lnPN = load i64*, i64** %cPp
108 %lnPP = getelementptr inbound %lnPP = getelementptr inbound
109 store i64 %lnPP, i64* %lnPP, lt store i64 %lnPP, i64* %lnPP, lt
110 %lnPR = load i64, i64* %lnPP %lnPR = load i64, i64* %lnPP
111 %lnPQ = load i64*, i64* %lnPP %lnPQ = load i64*, i64* %lnPP
112 %lnPS = getelementptr inbound %lnPS = getelementptr inbound
113 store i64 %lnPS, i64* %lnPP, lt store i64 %lnPS, i64* %lnPP, lt
114 %lnPT = ptrtoint %ssPk_closure %lnPT = ptrtoint %ssPk_closure
115 %lnPU = add i64 %lnPT, 1 %lnPU = add i64 %lnPT, 1
116 store i64 %lnPU, i64* %cPp %lnPU, i64* %cPp
117 %lnPV = ptrtoint i8* @base_G %lnPV = ptrtoint i8* @base_G
118 store i64 %lnPV, i64* %cPp %lnPV, i64* %cPp
119 %lnPW = ptrtoint i8* @base_S %lnPW = ptrtoint i8* @base_S
120 store i64 %lnPW, i64* %cPp %lnPW, i64* %cPp
121 %lnPX = load i64*, i64* %cPp %lnPX = load i64*, i64* %cPp
122 %lnPY = getelementptr inbound %lnPY = getelementptr inbound
123 %lnPZ = ptrtoint i8* @stg_ap_pp_fa %lnPZ = ptrtoint i8* @stg_ap_pp_fa
124 store i64* %lnPZ, i64** %cPp store i64* %lnPZ, i64** %cPp
125 %lnQ0 = inttoptr i64* %lnQ0, i64 125 %lnQ0 = inttoptr i64* %lnQ0, i64
126 %lnQ1 = bitcast i8* @stg_ap_p %lnQ1 = bitcast i8* @stg_ap_p
127 %lnQ2 = load i64*, i64* %cPp %lnQ2 = load i64*, i64* %cPp
128 %lnQ3 = load i64, i64* %cPp %lnQ3 = load i64, i64* %cPp
129 %lnQ4 = load i64*, i64* %cPp %lnQ4 = load i64*, i64* %cPp
130 %lnQ5 = load i64*, i64* %cPp %lnQ5 = load i64*, i64* %cPp
131 tail call ghcc void (i64*, i64* tail call ghcc void (i64*, i64*
132 ret void ret void 132 ret void
133 cPq: ret void 133 cPq: ret void
134 %lnQ6 = load i64, i64* %cPp %lnQ6 = load i64, i64* %cPp
135 %lnQ7 = inttoptr i64 %lnQ6 to i %lnQ7 = inttoptr i64 %lnQ6 to i
136 %lnQ8 = load i64, i64* %cPp %lnQ8 = load i64, i64* %cPp
137 %lnQ9 = inttoptr i64 %lnQ8 to %lnQ9 = inttoptr i64 %lnQ8 to
138 %lnQa = load i64*, i64* %cPp %lnQa = load i64*, i64* %cPp
139 %lnQb = load i64, i64* %cPp %lnQb = load i64, i64* %cPp
140 tail call ghcc void (i64*, i64* tail call ghcc void (i64*, i64*
141 ret void ret void 141 ret void
142 cPs: ret void 142 cPs: ret void
143 %lnQc = load i64, i64* %cPp %lnQc = load i64, i64* %cPp
144 store i64 %lnQc, i64* %cPp store i64 %lnQc, i64* %cPp
145 %lnQd = getelementptr inbound %lnQd = getelementptr inbound
146 %lnQe = bitcast i64* %lnQd to %lnQe = bitcast i64* %lnQd to
Second Experiment
Second Experiment

ppMatchedprogram_hide

196  %D6_Var = alloca double, i32 1
197  store double undef, double* %D6_Var
198  %cQm = alloca i64, i32 1
199  %R2_Var = alloca i64, i32 1
200  store i64 undef, i64* %R2_Var
201  %R1_Var = alloca i64, i32 1
202  store i64 %R1_Arg, i64* %R1_Var
203  %Sp_Var = alloca i64*, i32 1
204  store i64* %Sp_Arg, i64** %Sp_Arg
205  br label %cQp
206  cQp:
207  %lnQQu = load i64, i64* %lcQm
208  store i64 %lnQH, i64* %lcQm
209  %lnQH = ptrtoint i8* %lnQG to
210  %lnQG = call ccc i8* (i8*, i8*
211  store double undef, double* %D6_Var
212  store float undef, float* %F6_Var
213  store double undef, double* %D6_Var
214  store float undef, float* %F6_Var
215  br i1 %lnQA, label %cQp, labe
216  cQp:
217  %lnQQu = load i64, i64* %R1_Var
218  %lnQQu = load i64, i64* %R1_Var
219  %lnQQu = load i64, i64* %R1_Var
220  %lnQQu = load i64, i64* %R1_Var
221  %lnQQu = load i64, i64* %R1_Var
222  %lnQQu = load i64, i64* %R1_Var
223  %lnQQu = load i64, i64* %R1_Var
224  %lnQQu = load i64, i64* %R1_Var
225  %lnQQu = load i64, i64* %R1_Var
226  %lnQQu = load i64, i64* %R1_Var
227  %lnQQu = load i64, i64* %R1_Var
228  %lnQQu = load i64, i64* %R1_Var
229  %lnQQu = load i64, i64* %R1_Var
230  %lnQQu = load i64, i64* %R1_Var
231  %lnQQu = load i64, i64* %R1_Var
232  %lnQQu = load i64, i64* %R1_Var
233  %lnQQu = load i64, i64* %R1_Var
234  %lnQQu = load i64, i64* %R1_Var
235  %lnQQu = load i64, i64* %R1_Var
236  %lnQQu = load i64, i64* %R1_Var
237  %lnQQu = load i64, i64* %R1_Var
238  %lnQQu = load i64, i64* %R1_Var
239  %lnQQu = load i64, i64* %R1_Var
240  %lnQQu = load i64, i64* %R1_Var
241  %lnQQu = load i64, i64* %R1_Var
242  %lnQQu = load i64, i64* %R1_Var
243  br i1 %lnQJ, label %cQo, labe
244  cQn:
ppMatchedprogram_hide

245 \%lnQL = ptrtoint i8* @stg_bh_u  \%lnQL = ptrtoint i8* @ 245### \%lnQL = ptrtoint i8* @stg_bh_u
246 \%lnQK = load i64*, i64* \%Sp_ \%lnQK = load i64*, i64### \%lnQK = load i64*, i64* \%Sp_
247 \%lnQM = getelementptr inbounds store i64 \%lnQL, i64* \%lnQM, !
248 store i64 \%lnQL, i64* \%lnQM, !
249 \%lnQQ = load i64, i64* \%Sp_ \%lnQQ = load i64, i64### \%lnQQ = load i64, i64* \%Sp_
250 \%lnQN = load i64*, i64* \%Sp_ \%lnQN = load i64*, i6### \%lnQN = load i64*, i64* \%Sp_
251 \%lnQP = getelementptr inbounds store i64 \%lnQK, i64* \%lnQP, !
252 store i64 \%lnQK, i64* \%lnQP, !
253 \%lnQQ = ptrtoint \%Main_main_ store i64 \%lnQQ, i64* \%R2_Va
254 store i64 \%lnQQ, i64* \%Sp_ store i64 \%lnQQ, i64* \%R2_Va
255 \%lnQR = ptrtoint i8* store i64 \%lnQR, i64* \%base_G
256 store i64 \%lnQR, i64* \%R1_Va store i64 \%lnQR, i64* \%base_G
257 \%lnQS = load i64*, i64* \%Sp_ store i64 \%lnQS = load i64*, i64* \%Sp_
258 \%lnQT = getelementptr inbounds store i64 \%lnQR, i64* \%lnQT, !
259 \%lnQU = ptrtoint i64* \%lnQT to store i64 \%lnQU = ptrtoint i64* \%lnQT to
260 \%lnOV = inttoptr i64 \%lnQQ to store i64 \%lnOV = inttoptr i64 \%lnQQ to
261 store i64* \%lnOV, i64** \%Sp_V store i64* \%lnOV, i64** \%Sp_V
262 \%lnOW = bitcast i8* @store ap_p store i64* \%lnOV, i64** \%Sp_V
263 \%lnOX = load i64*, i64* \%Sp_ store i64* \%lnOX = load i64*, i64### \%lnOX = load i64*, i64** \%Sp_
264 \%lnOY = load i64, i64* \%R1_Vi store i64 \%lnOY = load i64, i64* \%R1_Vi
265 \%lnOZ = load i64, i64* \%R2_Vi store i64 \%lnOZ = load i64, i64* \%R2_Vi
266 tail call ghccc void (i64*, i64) tail call ghccc void (i64*, i64)
267 \%cQo: ret void ret void 267### ret void
268 \%lnRO = load i64, i64* \%R01D \%lnRO = load i64, i64* \%R01D
269 \%lnR1 = inttoptr i64 \%lnR0 to i \%lnR1 = inttoptr i64 \%lnR0 to i
270 \%lnR2 = load i64, i64* \%R1_R1, ! \%lnR2 = load i64, i64* \%R1_R1, !
271 \%lnR3 = inttoptr i64 \%lnR2 to v \%lnR3 = inttoptr i64 \%lnR2 to v
272 \%lnR4 = load i64*, i64* \%Sp_ \%lnR4 = load i64*, i6### \%lnR4 = load i64*, i64** \%Sp_
273 \%lnR5 = load i64, i64* \%R1_Ve \%lnR5 = load i64, i64### \%lnR5 = load i64, i64* \%R1_Ve
274 tail call ghccc void (i64*, i64) tail call ghccc void (i64*, i64)
275 ret void ret void 276### ret void
276 \%cQq: \%cQq:
277 \%lnR6 = load i64, i64* \%R01D \%lnR6 = load i64, i64* \%R01D
278 store i64 \%lnR6, i64* \%R1_Ve store i64 \%lnR6, i64* \%R1_Ve
279 \%lnR7 = getelementptr inbounds store i64 \%lnR7 = getelementptr inbounds
280 store i64 \%lnR7 = getelementptr inbounds store i64 \%lnR7 = getelementptr inbounds
281 \%lnR8 = bitcast i64* \%lnR7 to i store i64 \%lnR8 = bitcast i64* \%lnR7 to i
282 \%lnR9 = load i64, i64* \%R08, ! store i64 \%lnR9 = load i64, i64* \%R08, !
283 \%lnRa = inttoptr i64 \%lnR9 to v store i64 \%lnRa = inttoptr i64 \%lnR9 to v
284 \%lnRb = load i64*, i64* \%Sp_ store i64 \%lnRb = load i64*, i6### \%lnRb = load i64*, i64** \%Sp_
285 \%lnRc = load i64, i64* \%R1_Ve store i64 \%lnRc = load i64, i64### \%lnRc = load i64, i64* \%R1_Ve
286 tail call ghccc void (i64*, i64) tail call ghccc void (i64*, i64)
287 ret void ret void 287### ret void
288 \%cQq: \%cQq:
289 \%P2_bytes_struct = type <[5 x \%row_bytes_struct = type <[5:
290 @P2_bytes$def = internal cons \%row_bytes$def = internal con
291 @P2_bytes = internal alias i8, \%row_bytes = internal alias i8,
292 @Pb_closure_struct = type <i6- \%Pb_closure_struct = type <i6-
293 @Pb_closure$def = internal glo \%Pb_closure$def = internal glo

Page 6
ppMatchedprogram_hide

294 @rPb_closure = internal alias i8
295 %rPc_bytes_struct = type <[5 x i8>
296 @rPc_bytes$def = internal constan
297 @rPc_bytes = internal alias i8,
298 @rPd_closure_struct = type <[6 x i8>
299 @rPd_closure$def = internal glo
300 @rPd_closure = internal alias i8
301 @Main_zdtrModule_closure_struct
302 @Main_zdtrModule_closure$del
303 @Main_zdtrModule_closure = al
304 @integerzmwiredzmin_GHCziln
305 @stg_SRT_3_info = external glc
306 @base_SystemziIO_print_closu
307 @base_GHCziShow_zdfShowIn
308 @newCAF = external global i8
309 @stg_bh_upd_frame_info = ext
310 @stg_ap_pp_fast = external glo
311 @stg_SRT_2_info = external glc
312 @base_GHCziTopHandler_runMain
313 @stg_ap_p_fast = external glo
314 @ghczmprim_GHCziTypes_TrName
315 @ghczmprim_GHCziTypes_Module
316 @llvm.used = appending consta

### @rPb_closure = internal alias i8
### %rPc_bytes_struct = type <[5 x i8>
### @rPc_bytes$def = internal constan
### @rPc_bytes = internal alias i8,
### @rPd_closure_struct = type <[6 x i8>
### @rPd_closure$def = internal glo
### @rPd_closure = internal alias i8
### @Main_zdtrModule_closure_struct
### @Main_zdtrModule_closure$del
### @Main_zdtrModule_closure = al
### @integerzmwiredzmin_GHCziln
### @stg_SRT_3_info = external glc
### @base_SystemziIO_print_closu
### @base_GHCziShow_zdfShowIn
### @newCAF = external global i8
### @stg_bh_upd_frame_info = ext
### @stg_ap_pp_fast = external glo
### @stg_ap_p_fast = external glo
### @ghczmprim_GHCziTypes_TrName
### @ghczmprim_GHCziTypes_Module
### @llvm.used = appending consta