Synthesizing a Synthesis tool

by

Rohit Singh

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
Aug 30, 2013

Certified by .....
Armando Solar-Lezama
Associate Professor
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Synthesizing a Synthesis tool

by

Rohit Singh

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 30, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

SMT/SAT solvers are used by many tools for program verification and analysis. Most of these tools have an optimization layer which applies transformations (or "rewrite rules") to simplify the internal representation of the problem. These hard coded rules can drastically affect the performance of the solver. They are usually hand-picked by experts and verified using trial and error. These rules are very domain-specific as well (the domain from which the tool receives its inputs) and leverage the recurring patterns in the domain. The goal of this thesis is to automatically synthesize this optimization layer by learning an optimal set of rules from a corpus of problems taken from a given domain.

To achieve this goal, we will use two key technologies: Machine Learning and Program Synthesis (**Sketch** tool). **Sketch** is a state of the art tool for generating programs automatically from high level specifications. We propose a Machine Learning and **Sketch** based method to automatically generate "statistically significant" optimization rules (rules that can be applied at significant number of places in benchmarks from a particular domain) and then generate efficient code to apply the rules for that particular domain.

In addition to using **Sketch** as a tool, we will also use it as a target for this technology. **Sketch** uses SAT/SMT solver in its back-end, and, like any other tool it has its own hand-built optimization layer. In particular, **Sketch** uses a set of predetermined optimization rules to modify the internal representation of the formula in a way that results in faster SAT/SMT solving. **Sketch** is being used for synthesizing programs in various domains like Storyboard Programming[21], SQL queries for databases[5], MPI based Parallel Programming, Autograder for MOOCs[20]; The current optimizer has to work well for each one of these domains and one of our goals is to have a domain specific optimizer that can take advantage of specific features of each domain. Hence, **Sketch** is an ideal use-case for our analysis and all our experiments are conducted on the **Sketch** tool for various domains.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor

Acknowledgments

I would like to thank my adviser Prof. Armando Solar-Lezama, whose guidance was the most important ingredient for success of this project. Without his constant support and encouragement this thesis would not have been where it is now.

I would also like to thank my research group-mates, especially, Zhilei Xu and Rishabh Singh, who have always given the most useful and constructive feedback, support and have also helped me at various stages of this project.

One of the key components of this thesis which uses Machine Learning, was done as a project in the Advanced Machine Learning class (6.867) under the guidance of Prof. Leslie Kaelbling and Prof. Tomas Lozano-Perez. A special thanks to them as well for letting me pursue this project and help me chose the right tools.

Last but not the least, I would like to thank my family and friends for always being there for me and supporting me at every stage of my life.

Contents

List of Figures

Chapter 1

Introduction

Existing SAT solvers can handle problems with millions of clauses and variables that are encountered in various verification and synthesis problems. SAT solving has thus become a major tool in automated analysis of software and hardware. **Sketch** is a state of the art tool for generating programs automatically from high level specifications. **Sketch** uses internal DAGs (Directed Acyclic Graphs) to represent partial programs which is then translated into constraints for the SAT solver. It also uses a set of pre-determined optimization rules for modifying the DAG representation in a way that results into faster SAT solving. These rules have been hand-picked by experts and verified using trial and error methods. At the same time, **Sketch** is being used for synthesizing programs in various domains like Storyboard Programming[21], SQL queries for databases[5], MPI based parallel programming, Autograder for MOOCs[20] etc. The problems from various domains tend to have different properties and patterns which makes the synthesis process slow because of these hard-coded rules which are kept the same across domains. We propose a Machine Learning and **Sketch** based method to automatically generate "statistically significant" optimization rules (rules that can be applied at significant number of places in benchmarks from a particular domain) and then select the best rules for a particular domain. The following sections will set up the exact problem with details.

1.1 Sketch : A program synthesis tool

Sketch [23] is a state of the art tool for generating programs from specifications which are given as partial C programs. **Sketch** specifications contain special "holes" or "controls" represented by two consecutive question marks "??". These "holes" represent an arbitrary integer and the purpose of **Sketch** tool is to instantiate these "holes" with fixed integer values (from a bounded set usually defined by number of bits to be used for representing the hole value) so that all assertions and constraints specified in the program are satisfied. Figure 1-1 shows a simple **Sketch** program where the partial program with a hole is completed so that the given specification (in the form of an assertion on the output value of the function) is satisfied. Sketch supports most of the C-like operations (including arrays and structs) and also has some other constructs which enable us to specify requirements effectively.

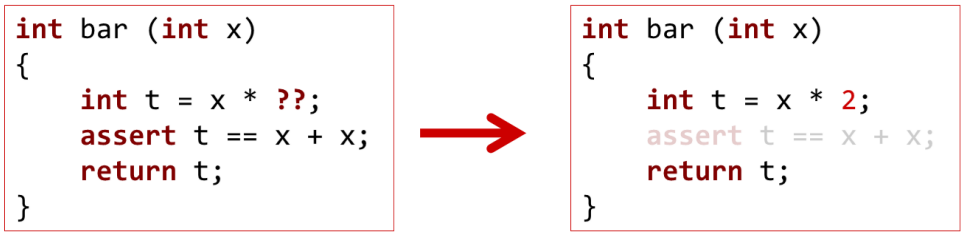


Figure 1-1: A **Sketch** program completion example

1.1.1 Generators

Another language construct in **Sketch** that is relevant to this thesis is the "generators" [23]. A **generator** represents a mechanism in sketch to define a set of code fragments. In particular, it can be used to give recursive specifications of a grammar of expressions. For example, Figure 1-2 represents a grammar of expressions formed with bit-vectors and bit-vector operations. Using generators, we can represent grammars in an elegant and concise way. Later on we will use generators in **Sketch** to represent grammar of **Sketch** 's internal representation.

```
generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return !gen(x, bnd-1);
    if(??){
        return {| gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) |};
    }
}
```

Figure 1-2: A sample **Sketch** generator representing the grammar of Boolean expressions formed with bit-vectors

1.1.2 *Minimize* construct

Sometimes there are multiple candidate solutions (programs) which satisfy the given **Sketch** specification and we would like to get the "best" one out of them. **Sketch** provides a mechanism to optimize a simple user defined metric. One can use the "minimize" keyword [20] in **Sketch** to inform the tool that out of all possible candidate solutions it should chose the one which minimizes this expression provided as a parameter to "minimize".

```
generator bit[W] gen(bit[W] x, int bnd, ref int cost){
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) {cost++; return !gen(x, bnd-1);}
    if(??){
        cost+=2;
        return {| gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) |};
    }
}

... main(...){
    assert(gen(x,4,cost) == res(x));
    minimize(cost);
}
```

Figure 1-3: A **Sketch** generator with a cost parameter to be minimized

1.1.3 CEGIS

Sketch uses **Counter-Example Guided Inductive Synthesis** [22] to solve these partial programs and come up with a candidate program that satisfies the given specification (in the form of a partial program with constraints). Abstractly, the synthesis problem in **Sketch** can be seen as solving a second order formula of the form shown in (1.1). Where, c represents the "controls" or "holes" present in the partial program (called **sketch** in the equation), x is the vector of all inputs to the **sketch** and **spec** represents a specification for the **sketch** which may or may not be complete and may have some assertions inside it.

$$\exists c. \forall x. \mathbf{spec}(x) = \mathbf{sketch}(x, c) \tag{1.1}$$

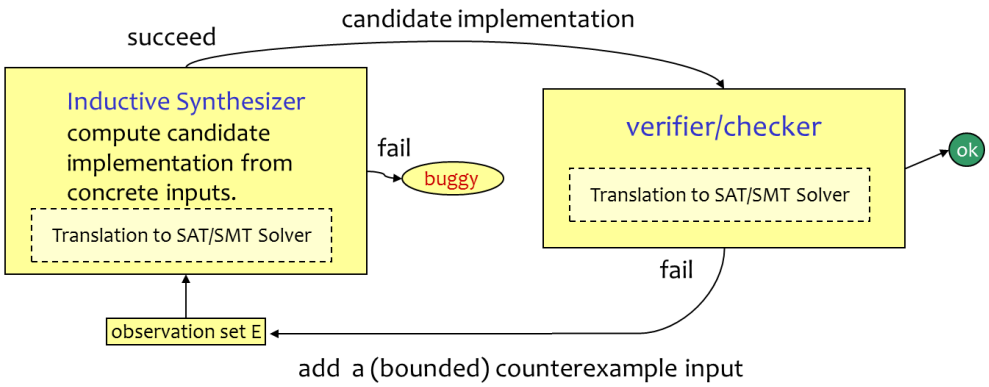


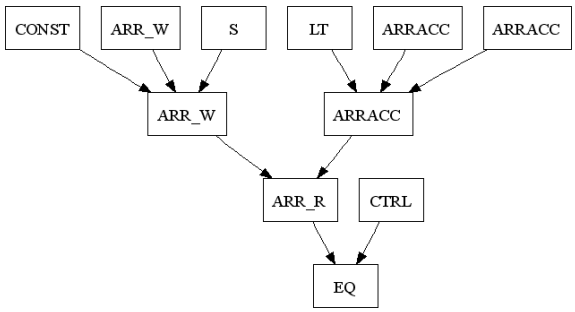
Figure 1-4: CEGIS algorithm used in **Sketch**

CEGIS algorithm (shown in figure 1-4) guesses a candidate set of values for c and then gives it to the verifier to check if this value for c satisfies the specification. If it works, then we have what we wanted: a candidate program which satisfies the specification. If not, the verifier produces a counter-example set of values for x and then the tool uses inductive synthesis to try and prune out candidate programs based on the counter-example set and the cycle continues until we find a correct solution.

Sketch bounds the number of candidate programs by bounding the number of bits for each "control" variable which guarantees the termination of this procedure.

One important thing to note here is that both the "Inductive Synthesis" and "Verification" phases translate their respective problems to SAT/SMT and use a solver for doing their job. This is where we will be focusing in this thesis report.

1.1.4 Problem transmission to the SAT/SMT Solver




```
id = ARR_R      TYPE    index   inputarr
id = ARR_W      TYPE    index   old-array       new-value
id = ARR_CREATE TYPE    size    v0 v1 ....
id = BINOP      TYPE    left    right
        // where BINOP can be AND, OR, XOR, PLUS, TIMES, DIV, MOD, LT, EQ
id = UNOP       TYPE    parent  // where UNOP can be NOT or NEG
id = SRC        TYPE    NAME    bits
id = CTRL       TYPE    NAME    bits
id = DST        TYPE    NAME    val
id = UFUN       TYPE    NAME    OUT_NAME CALLID ( (size p1 p2 ...) | (***) )
id = ARRACC     TYPE    index   size    v0 v1 ...
id = CONST      TYPE    val
id = ARRASS     TYPE    val == c noval yesval
id = ACTRL      TYPE    nbits b0 b1 b2 ...
id = ASSERT     val     "msg"
```

Figure 1-5: A DAG, Internal representation in **Sketch**

Sketch stores the problem (partial programs with assertions) in an internal representation which is a Directed Acyclic Graph. An example DAG and the complete grammar used for representing DAGs is shown in figure 1-5. This internal representation is expressive enough to describe these problems obtained from partial C-like programs and is similar to the representation used by other solvers. To minimize the amount of memory used by this representation and also the amount of time taken by the solver, **Sketch** does a lot of optimizations to make the size of this DAG as small as possible, at the same time, trying to ensure that the translation to SAT/SMT results info faster solving. This is done by using "rewrite rules" to simply the representation and then "encoding rules" to transform the problem to the language of the constraint solver (a SAT solver in the case of **Sketch**). The transformation process is shown in figure 1-6.

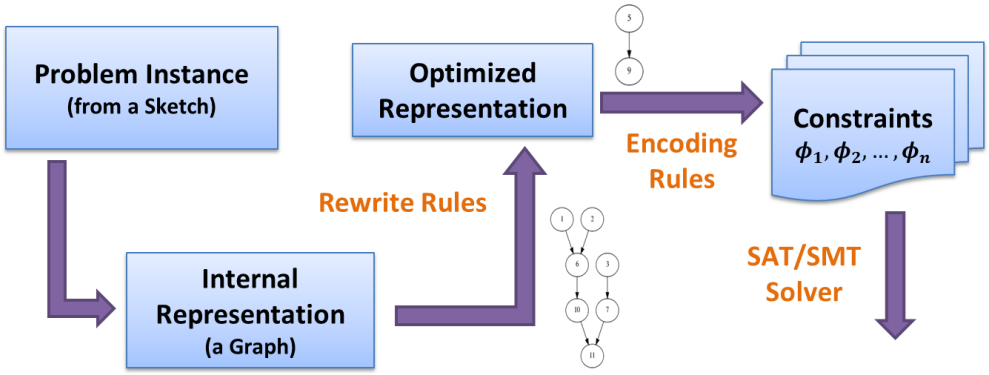


Figure 1-6: DAG transformation in **Sketch**

This whole transformation process is designed in a way so that its possible to introduce new internal representation (like adding support for arrays), new rewrite rules (to optimize new additions to internal representation) and new encoding rules (to make the best use of the solver) but its a tedious process given all the code one has to write and debug, and the huge set of possible rules one could use. The state of the art methodology to achieve this is via trial and error along with expert knowledge. The system developer tries a particular chain and if its not of much use or makes the solver slow, he has to go back and make changes to some parts of the pipeline or come up with a new one.

Our aim in this thesis is to help the tool developer by automatically generating efficient code for a part of this pipeline so that they can focus on the other features. We will focus on the "rewrite rules" aspect of this pipeline in this thesis work (we will explain "rewrite rules" and their properties in detail later). The idea is to automate the process so that the rules can be generated automatically through a combination of Synthesis and Machine Learning.

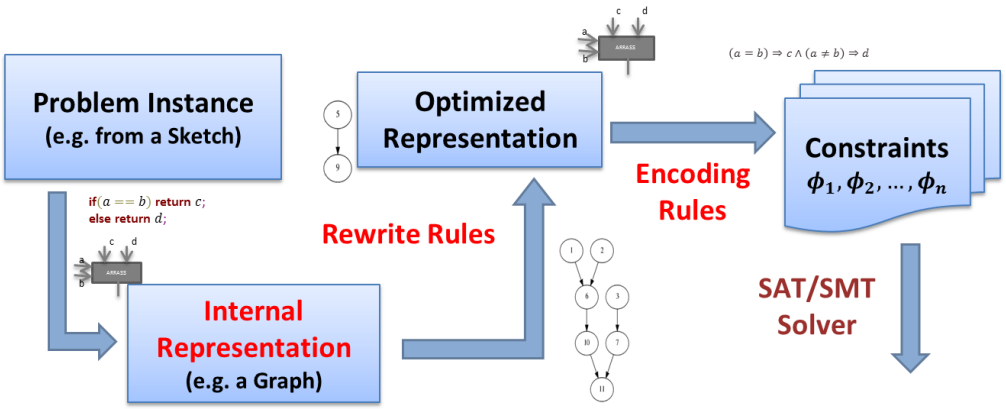


Figure 1-7: General transformation pipeline in many tools employing SAT/SMT solvers

1.2 Solvers and Tools employing solvers

The picture shown in figure 1-6 is not unique to **Sketch**. In fact one can generalize it to many tools (figure 1-7) which are either themselves solvers or employ solvers in the back-end. Some such examples are KINT [24], Jeeves [27], BBR [4], UCLID [14], Z3 [6] etc. Every tool which uses solvers has a pipeline which looks like this embedded somewhere in the system. Although, the overall idea and structure is the same across tools, they differ in terms of the choices they make for:

1. Internal Representation
2. Rewrite Rules
3. Encoding Rules

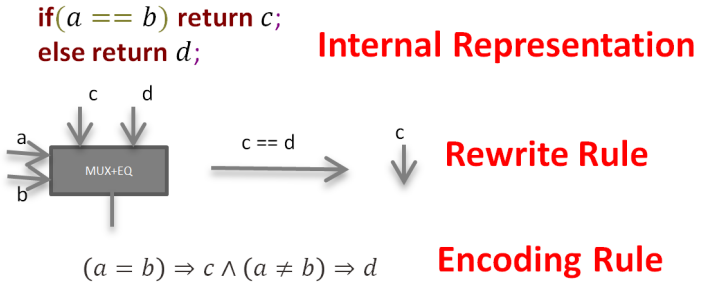


Figure 1-8: Example transformation pipeline

These choices are very domain specific and hence the tools perform well in different domains. To give an example of decisions one needs to make, lets consider the example shown in figure 1-8. The given block of code has 4 inputs a, b, c, d and the tool developer can chose to represent it traditionally using a MUX and EQ separately or he can merge both of them into one node as shown in the example. Now, after being merged into one node, we need to come up with new rewrite rules for whenever a sub-graph contains this node. One such simple rewrite rule is also shown in the figure 1-8 which replaces the whole node with just one input variable (c) in case the inputs satisfy a predicate ($c == d$). And, finally, the tool developer has to provide an encoding rule to translate this into a constraint for the solver. One such encoding is also provided in the figure 1-8. These rules can depend on the source from which the input comes to this node and thus can be very tedious to find and code/debug manually. This is where our work comes in and eradicates the need for the developers to think about the rewrite rules.

1.3 Rewrite rules

Rewrite rules are simple transformations which can be applied under certain assumptions which have to be checked before applying the rule. Every rewrite rule has three components: the left-hand-side (**LHS**), the predicate **Pred** (in the center) and the right-hand-side (**RHS**). All three of them are functions of the inputs (x).

Definition 1. A **Rewrite Rule** is a triple ($\mathbf{LHS}\ (x), \mathbf{Pred}\ (x), \mathbf{RHS}\ (x)$) where x is a vector of input variables from a domain D and **LHS** , **RHS** are functions whose structure is defined by a grammar and **Pred** is a predicate defined over the same input variables from a different grammar such that it satisfies the following formula:

$$\forall x.\mathbf{Pred}\ (x) \implies (\mathbf{LHS}\ (x) = \mathbf{RHS}\ (x))$$

It can be interpreted as saying that whenever the predicate **Pred** shown in the center is satisfied by the inputs, one can replace such an occurrence of **LHS** in the internal representation (e.g. DAG) by **RHS** which will potentially reduce the size of the representation and eventually the size of the formula sent to the solver. Some examples are shown in figure 1-9.

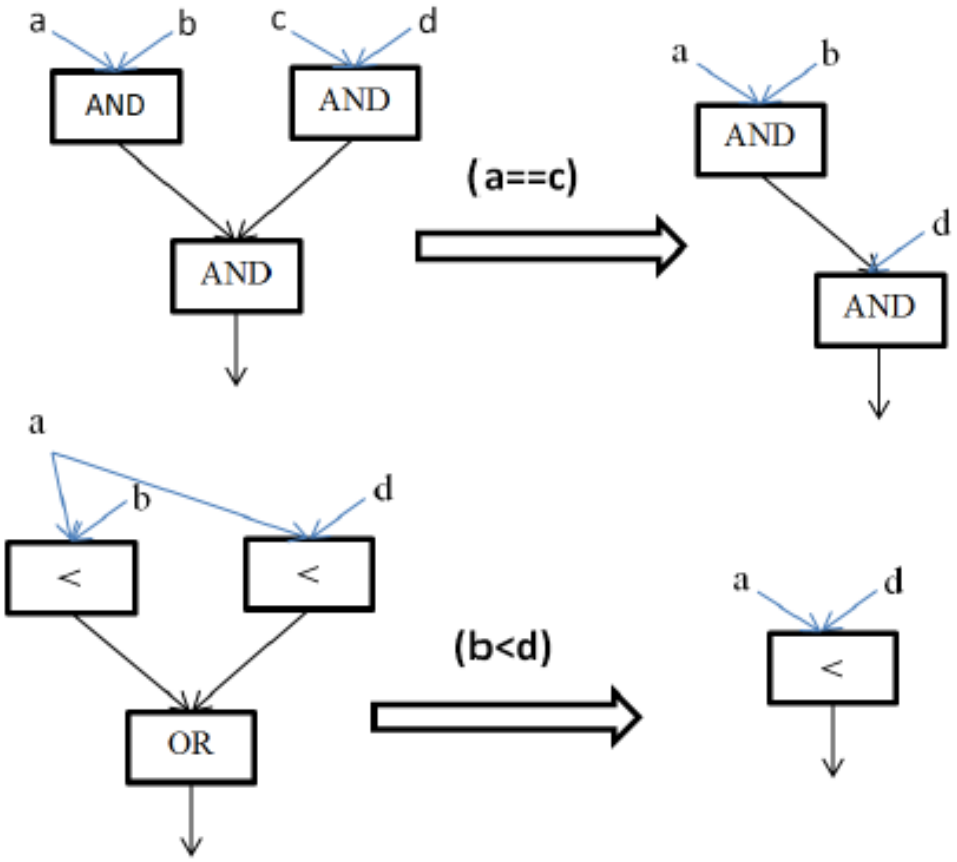


Figure 1-9: Some examples of Rewrite Rules

In the examples, each node represents an operation and the inputs are presented by incoming arrows. In words, the first rule can be seen as: "If $(a == c)$ then we can replace $(a \wedge b) \wedge (c \wedge d)$ with $(a \wedge b) \wedge d$". Its a simple rule but it can reduce the size of the problem by a lot if the **LHS** pattern occurs often in the domain from where the problems are being sourced.

Ideally we want the **RHS** to be of smaller size than **LHS** so that it can help us

reduce the overall problem size. Here we make an inherent assumption that making local changes can lead to global effects in the same direction. This has been verified empirically in **Sketch** as well. There are some examples from **Sketch** back-end in figure 1-10.

1.3.1 Challenges

In this sub-section we will try to list all the challenges one faces while trying to deal with the paradigm of rewrite rules.

Coming up with the rules

One needs to have a deep understanding of the domain from which the problems are being fed to the tool. At the same time, the same person should know enough about the internal representation and working of the tool so as to come up with good candidate rules. If the **LHS** is complicated enough, checking correctness of the rule can also be troublesome.

As can be seen in figure 1-10 the rules are fairly complicated and its only after one implements and applies a particular rule, they see more patterns in the problems and realize that some extensions of the rule can capture them as well. The rules shown in figure 1-10 are related to each other in that sense and going from one to another is still a lot of effort.

Coding up the rules

The tool developer has to trade-off performance versus readability. If he favors performance, then its easy for him to introduce bugs and also make it difficult for a new developer to understand what is going on in the code. On the other hand, if all rules are separately added in a more readable format, the tool will have to consider the rules independently which will disable opportunities for some optimizations that

could have been done by sharing pattern matching costs across the rules.

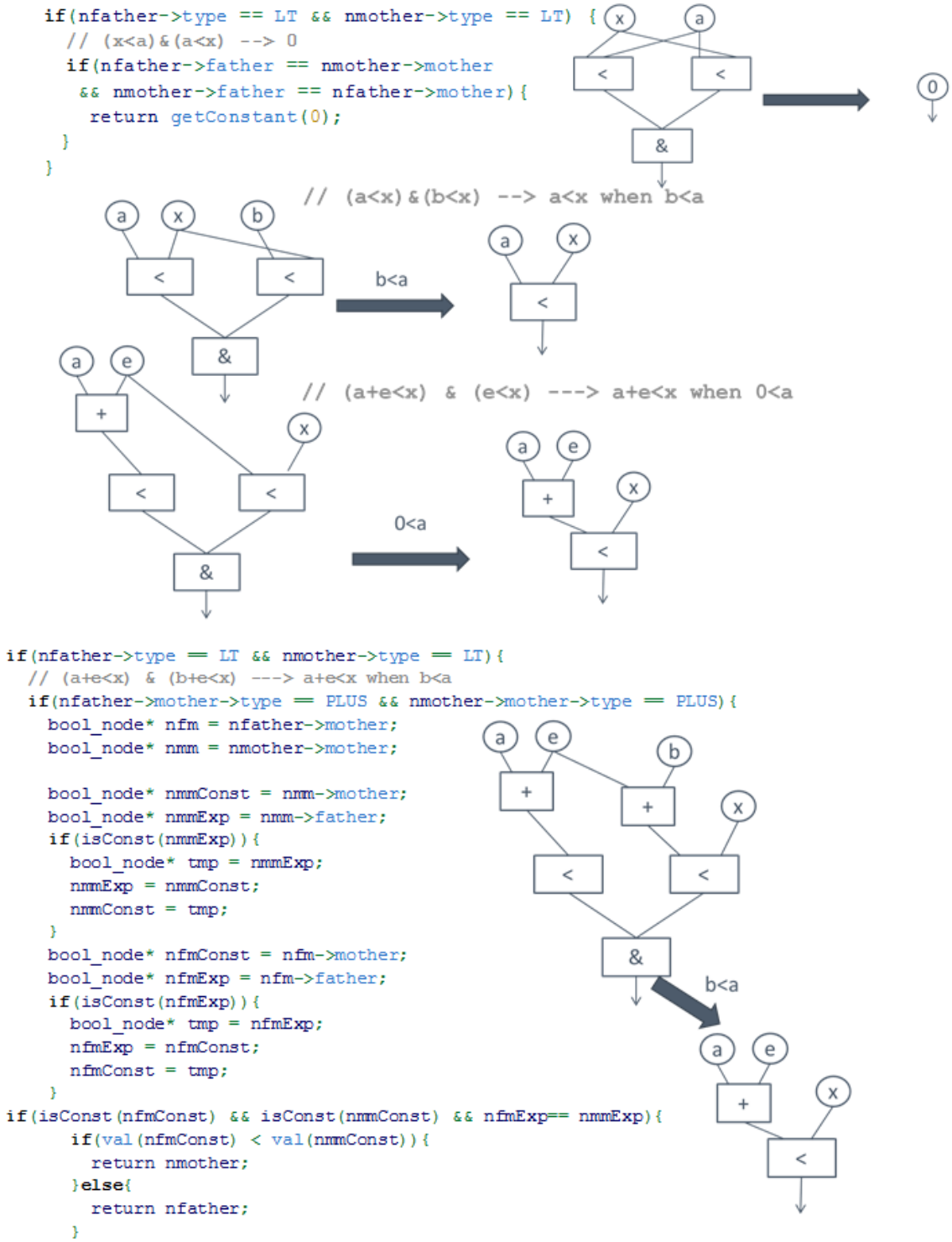


Figure 1-10: Example Rewrite Rules from **Sketch**

Another issue with coding up the rules is with the "symmetries" of the rule. If there are some commutative operations, the developer has to take care of each case of predicate evaluation with permuted inputs. He may forget to add a case and that can result into wrong validation.

As can be seen in figure 1-10 the code for the last rule is fairly complicated and has to take into account various symmetries of the "plus" operation. The rules are built incrementally to avoid the bad effects of one rule being applied before other. Finding the right order of application of rules is a challenge as well.

Trial and validation

There's no other way to guarantee performance except for trying it out. It can be a long and tedious cycle and combined with errors introduced while coding up the rules, it can take years to find the best set of rewrite rules that work well.

The rules in figure 1-10 are hand-crafted but have been tested and validated on various problems from multiple domains. Some similar rules have been removed because their effects on size was not good enough as compared to the time it took to match them.

1.4 Impact of Rewrite Rules on Sketch

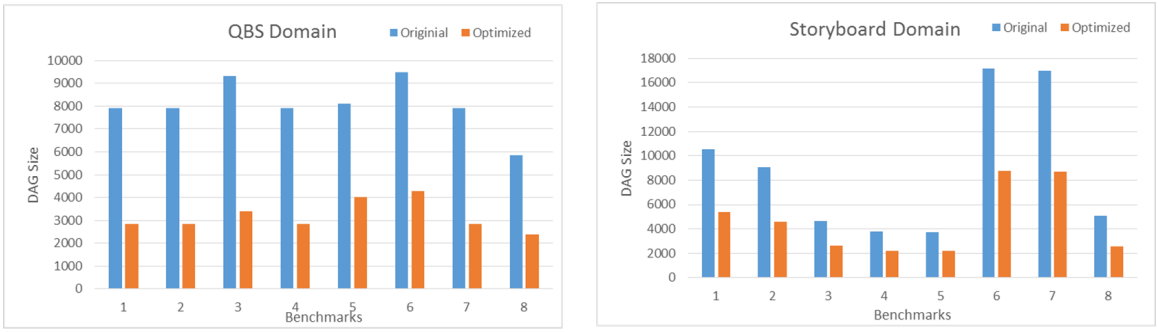


Figure 1-11: Impact of Rewrite Rules on **Sketch**

To show how important the rewrite rules are, we conducted an experiment where we removed all the rules present in Sketch. We measured the size of the problems generated at the beginning of the CEGIS process and compared it with the case when the rules were present. The times taken to solve the problem differed by a lot (minutes vs seconds and an hour vs minutes) due to the sizes of the problems . We present comparison of sizes of problems in figure 1-11 for the Storyboard programming [21] and QBS query synthesis [5] domains where we are using the same tool **Sketch** with and without the hand-written rewrite rules. There is approximately a $2x$ decrease in size which might have a disproportionate effect on time complexity (since most SAT/SMT solving algorithms are exponential in size of the problem). And, this is what we observed with running time when we ran the experiments.

1.5 Problem Statement

Now that we understand the challenges one faces while working with rewrite rules, we want to make this process easier for the developer using Machine learning and Synthesis. We want to build a framework for automatically generating and efficiently implementing important "rewrite rules" for a given domain for Solvers and tools which employ solvers.

To be more specific, we are going to solve the following problem:

Problem 1. Automatically generate "rewrite rules" of the following structure:

$$\text{If } \textbf{Pred } (x) \text{ then } \textbf{ replace LHS } (x) \text{ with } \textbf{RHS } (x)$$

such that the following hold:

- **Correctness**: $\textbf{Pred } (x) \implies (\textbf{LHS } (x) = \textbf{RHS } (x))$
- **Relevant Rules**: the generated rules are "statistically significant" for a domain

(benchmark problems provided)

- **Code Generation**: the automatically generated code is efficient and correct

Chapter 2

Related Work

A Pre-processing step in Solvers and tools employing solvers (like Z3, **Sketch** etc) is an essential one and term rewriting has been extensively used as a part this pre-processing step [15]. It is hard to find papers describing the pre-processing steps used in modern SAT/SMT solvers and other tools which use them. Most SMT solver developers agree that these pre-processing steps are very important for the specific domains e.g. bit-vector theory or specific **Sketch** domains. We believe these techniques are not published for several reasons: most of them are little tricks that by themselves are not a significant scientific contribution; And, most of the techniques only work in the context of a particular system or domain; a technique that may seem to work very well with tool A, may not work with tool B. As a domain specific example, [26] focuses heavily on word-level simplifications like application of rewrite rules to avoid exponential blowup during construction of the problem (along with other techniques for optimization). This supports our claim that a lot of time is spent in finding simplification rules for each domain and automating the process will definitely aid the developers.

Each part of our tool-chain solves an independent problem and is different from the state of the art, specialized for our purposes. Identification of recurrent sub-graphs

from benchmark DAGs is similar in essence to the Motif discovery problem [18] which is famous because of its application in DNA fingerprinting[11]. This is a very active area of research and recently we have seen some attempts to use Machine learning [13] and distributed systems [16] to compute the Motifs (statistically significant recurrent sub-graphs) as quickly as possible. Our DAGs, on the other hand, have labeled nodes (labeled with operation types) which makes direct translation to Motif discovery problem much difficult. Also, to avoid using graph theoretic algorithms, we use Machine learning (clustering) based method to search for statistically significant sub-graphs.

In superoptimization community, people explore all possible equivalent programs and find the most optimal one. It would not make sense to do that for formulas. But [2] came up with this idea of packaging the superoptimization into multiple rewrite rules similar to what we are doing here. Although it looks similar in spirit to our work, there are a few differences. Most importantly, [2] uses enumeration of potential candidates for optimized versions of instruction sequences and then checks if it is indeed the most optimal version. Whereas, we use constraint based synthesis for generating the rules which offers a possibility of specifying a structured grammar for the functions.

The third phase which automatically generates optimizer's code (representing an abstract reduction system) is similar to a term or graph rewrite system like Stratego/XT [3][17] or GrGEN.NET [8]. Stratego/XT is a framework for the development of program transformation systems and GrGEN.NET is a Graph Rewrite Generator. They offer declarative languages for graph modeling, pattern matching, and rewriting. Both of these tools generate efficient code for program/graph transformation based on rule control logic provided by the user. We built up on their ideas and wrote our own compiler because we already had special input graphs (DAGs with operations as labels of nodes) and an existing framework for optimization (**Sketch** DAG optimiza-

tion class). Our strategy can be compared with LALR parser generation [9] where the next look-ahead symbol helps decide which rule to use. In our case as well we keep around a set of rules that are potentially applicable based on what the algorithm has seen.

Chapter 3

Overview of the Proposed Solution

In this chapter, we will present an overview of the proposed algorithm for automation and the technologies involved. Note that we will be using **Sketch** as both our target tool for optimization and also as a tool for synthesis of rules. This will fix the "internal representation" and "encoding rules" we will be dealing with. But the technique (and the implementation) can be extended to other Solvers or tools which use SAT/SMT solvers and have a pass similar to the one mentioned in section 1.2.

The input to our algorithm would be some DAGs obtained using **Sketch** with a particular domain's benchmarks. We will first try to find sub-graphs of fixed depth in these benchmark DAGs which occur more often than others and then we will try to find rules for these sub-graphs (rewrite rules which can simplify these sub-graphs). Then we will use our efficient code generator to provide us with an "optimization library" which we can plug back into **Sketch** and then chose a subset of rules based on the performance on these benchmarks. We can re-iterate the process to find more and more rules for the new patterns which are now available (the closed loop is shown in figure 3-1.

The automation consists of three major steps (figure 3-1):

1. Obtaining recurrent sub-problems/sub-graphs: Clustering based approach

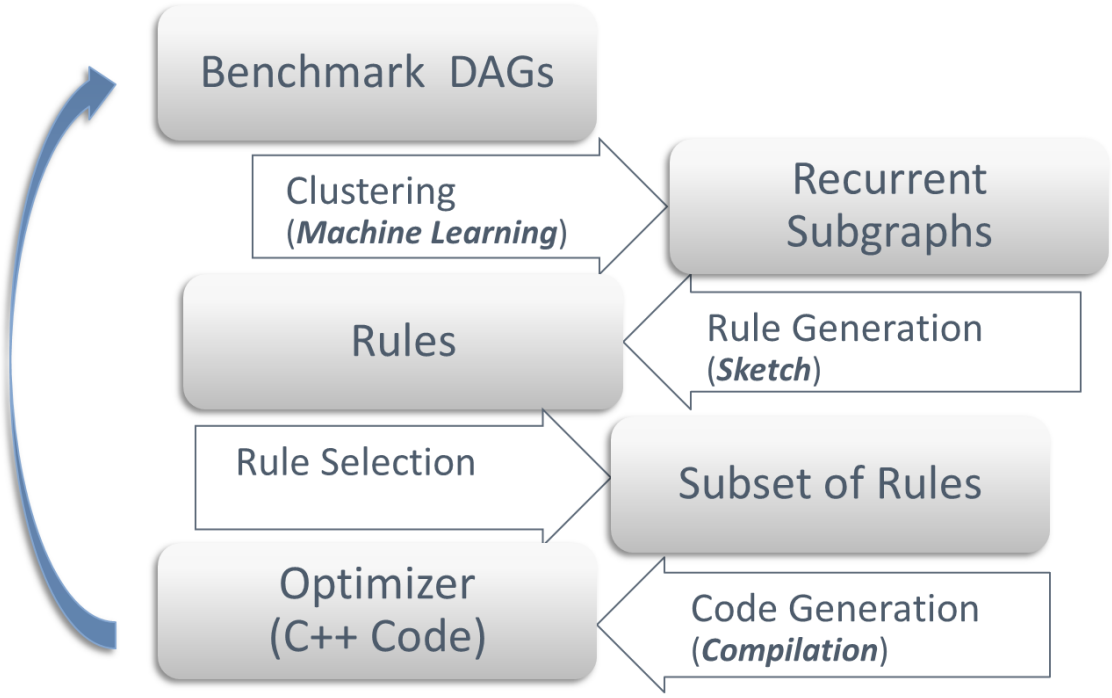


Figure 3-1: Overall steps for the automated solution

2. Automatically generating "correct" and "smallest" rewrite rules for the recurrent sub-graphs obtained in the previous step: **Sketch** based synthesis approach

3. Generating efficient code for pattern matching and rule application: Compilation of "relevant" rules

We will give a short description of the steps in this chapter.

3.1 Obtaining recurrent sub-graphs

The motive behind this phase of the algorithm is to prune the set of all possible rules by looking at only those which we hope might make a difference. We achieve this by fixing the **LHS** of the rule as one of the most-occuring patterns in the benchmark DAGs.

There is a mathematical characterization of the general problem of finding re-

current and statistically significant sub-graphs (called Motifs) in an arbitrary graph. There are well known graph theoretic Motif discovery algorithms but none of them scale well[18] for large graphs.

Our problem is a more specific and approximate version of the Morif discovery problem. We have some specific "Types" of nodes which should be preserved while finding sub-graphs. All practical algorithms for Motif discovery ignore the "Type" or color (in graph theoretic terms) of the nodes and hence will not produce good results for us. Also, they are fairly slow in general due to precision requirement. Since we want something that is approximately recurrent, we would be better off using an algorithm which gives us better insights in lesser time.

We solve this problem by using clustering based machine learning techniques. We also compare the results with traditional Motif discovery tools available on the internet and show why clustering is a better and faster way to solve our problem. More details will be provided in chapter 4.

3.2 Generating correct and effective rewrite rules

We receive the **LHS** of the rewrite rule from the clustering based search. Now, we need to find predicates **Pred** and **RHS** so that (**LHS** , **Pred** , **RHS**) is a valid rewrite rule (subsection 1). We also want to ensure that the **RHS** is of smaller size than **LHS** so that the local changes can potentially lead to global smaller sized problems.

The correctness and size constraints along with generation of the **Pred** and **RHS** is done using **Sketch** based program synthesis techniques (section 1.1). We use a grammar for **Pred** and **RHS** , and, write templates for them in the **Sketch** language using "holes" inside generators (subsection refsubsec:generators). Then, we write a **Sketch** specification which ensures validity of the rules given semantics of each node,

again, in **Sketch** as a library (Note that we had to model the way **Sketch** internal representation works using the external interface of the same **Sketch** tool). Running this **Sketch** specification would give us a rule that is correct and can be applied to replace the **LHS** under the assumptions given by **Pred** .

Although, this rule generated by **Sketch** would be correct, its not guaranteed to be "good". In fact, it may even just return an identity rule where **RHS** is the same as **LHS** . To avoid this problem and to ensure the size constraints we use the **minimize** keyword from the **Sketch** language (subsection 1.1.2). We associate a cost with **RHS** generator and minimize the size of the **RHS** and at the same time explicitly writing a constraint saying **size**(**LHS**) > **size**(**RHS**). This will ensure that not only we get a "good" rule that will reduce the size of the problem but also that for a given predicate it will be the "best" rule in the sense that no other smaller **RHS** would work for this **LHS** , **Pred** pair.

We use some predicate refinement based techniques to have a directed search of rules along with trying to get as many rules as possible. We will later on select a smaller subset of rules by observing which rules have the most impact on the benchmarks. Thus, we will have a set of rewrite rules generated automatically for a particular domain which work well for that domain.

3.3 Code generation: Compilation

Given a set of rules, we want to automatically generate correct code for pattern matching and replacement of sub-graphs. At the same time, we want to take benefit of the fact that since the tool developer doesn't even have to look at this code, we don't have readability constraints and hence we can try to optimize the code as much as possible and in the process, we may make it less understandable. One of the major optimizations that we perform is that we allow the rules to share pattern matching

code. The pattern matcher remembers the history of matching and tries to merge the **LHS**s of all the rules as much as possible so that even if a particular rule couldn't be applied, the pattern matching for the other rule will not cost much

Chapter 4

Statistically Significant Sub-graph Search: Clustering

Our aim in this chapter will be to solve the problem of finding "significant" and "recurrent" sub-graphs from a given set of benchmark DAGs. As mentioned earlier, a more concrete version of the problem is known as the Motif discovery problem and there are many state of the art algorithms for tackling it.

4.1 Motif discovery problem

Motif discovery became popular due to the problem of finding patterns in sequences of DNA. Researchers analyze and compare many DNA strands of equal length and find the most closely-matching sequences of a certain length in each strand. These patterns are of great scientific interest to those doing research in genetics because they correspond to sequences of DNA that control the activation of specific genes. Abstractly, A Network Motif is defined as a recurrent and statistically significant sub-graph or pattern of a larger graph or network. This definition can be stated in probabilistic terms as well [25]. There are many algorithms for finding Motifs in modern literature and it is currently a fairly active area of research. Finding "good"

motifs quickly and efficiently is very important. The state of the art algorithms can be classified as exact counting methods, sampling methods, pattern growth methods and so on. Recently there have been a push on using distributed systems to make the algorithms faster [1] [19].

4.2 Finding recurrent patterns in Sketch benchmark DAGs

Our problem of finding significant and recurrent sub-graphs in **Sketch** benchmark DAGs (from a particular domain) is different from simple Motif discovery problem because:

1. The type information of nodes (Nodes are typed as "AND", "OR" etc in **Sketch** benchmark DAGs) is not considered in the naive translation of our problem to Motif discovery problem. For our sub-graphs to be a part of rewrite rules, they need to have type information which is "recurrent" along with the structure.

2. The strict requirement of "significance" is not needed for our purposes and an approximately "significant" sub-graph might as well work.

4.3 Clustering based approach

Given the nature of the requirements, we decided to try out a clustering based approach. We assigned "feature vectors" to every node based on the local structure around it and then tried to find out nodes which are similar using a classifier which will cluster the similar ones together. Then we took a few samples from each class (ordered by the number of members in the class) and then moved on to generating rules for them. The process is outlined in figure 4-1. We will also use a graph theoretic

Motif discovery tool called Kavosh [12] and compare the results with the clustering based approach.

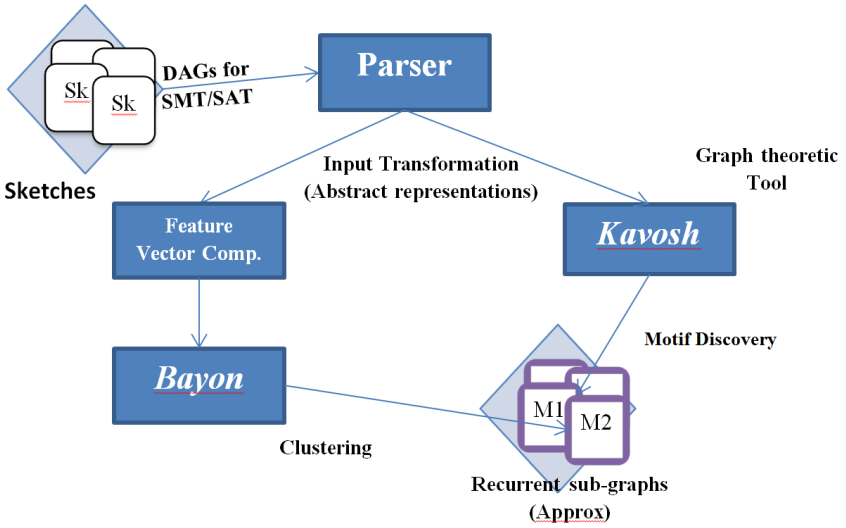


Figure 4-1: Diagram depicting the sub-graph search from **Sketch** benchmark DAGs

4.4 Sub-graph search framework

We obtain the DAGs from **Sketch** benchmarks for a particular domain. Then, we parse them to either feed them to the Kavosh [12] tool for finding Motifs or transform them into feature vectors for each node. We send the feature vectors to Bayon (a fast sparse feature vector based clustering tool [7]) and collect the results back in the form of recurrent sub-graphs. We explain each part of the chain separately as follows:

4.4.1 Input/Output specification

We obtain the DAGs for problems obtained from three different domains:

1. Services for a responsive system (labeled QBS) [5] and
2. Storyboard programming[21]
3. Autograder for MOOCs [20]

The problems correspond to synthesizing a system (filling the empty holes automatically) with the help **Sketch** [23]. The DAGs are obtained from the tool using a specific flag as input. The grammar for internal representation which is relevant to us is displayed in figure 1-5 (Right hand side) and Table 4.1. `id` represents the node identifier in the graph and different types of nodes have different labeled operation (`TYPE`) and potentially even different number of parent nodes. The parent nodes are mentioned after `TYPE`. The relevant node `TYPE`s (or the operations they represents) have been classified as:

1. `ARR_R` : Array read operation
2. `ARR_W` : Array write operation
3. `UNOP` : Unary operations (NOT, Unary MINUS)
4. `BINOP` : Binary operations (AND, OR, PLUS etc.)
5. `S/CTRL/CONST`: Nodes with no parents (Inputs, "holes" and constants)
6. `ARRACC`: Multiplexer nodes
7. `ARRASS`: Multiplexer + EQ (composite node)
8. `ASSERT`: Assertions (only one parent)

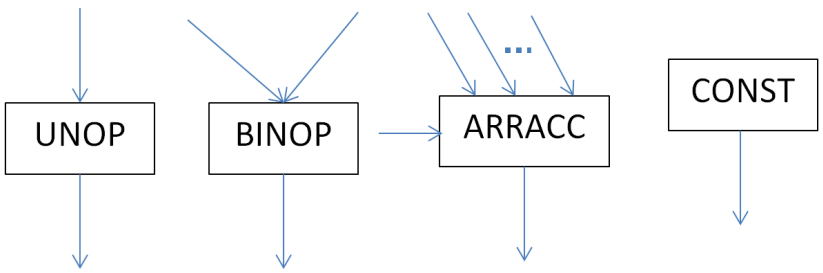


Figure 4-2: **Sketch** DAG nodes visualization: parents/inputs, children/outputs

Table 4.1: **Sketch** DAG node specification grammar

ID = ARR_R TYPE **index inputarr**
ID = ARR_W TYPE **index oldarray newvalue**
ID = BINOP TYPE **left right**
// BINOP CAN BE AND, OR, XOR, PLUS, TIMES, DIV, LT, EQ, MOD
ID = UNOP TYPE **parent** // UNOP CAN BE NOT, NEG
ID = S TYPE NAME BITS
ID = CTRL TYPE NAME BITS
ID = ARRACC TYPE **index** SIZE **v0 v1** ...
ID = CONST TYPE **val**
ID = ARRASS TYPE **val** == C **noval yesval**
ID = ASSERT **val** "MSG"

We represent the nodes with boxes (or circles) and label them by their `TYPE`. Some of these nodes with labels are shown in figure 4-2.

We transform this input DAG to a nascent abstract graph with no labels for trying out the graph theoretic Motif discovery algorithm and compute the feature vectors for the other case when we want to use the clustering algorithm.

4.4.2 Benchmarks

We will be using the DAGs obtained from three different domains: QBS, Autograder and Storyboard. Each benchmark consists of $14 - 20$ DAGs each comprising of $3000 - 10000$ nodes. We merge the DAGs obtained from all benchmark in a domain to form a huge DAG for the domain. The combined number of nodes present in the data-sets is between $200,000$ to $600,000$ based on whether we use the previously existing optimizations or not.

4.4.3 Tools and algorithms used

The important boxes in the system design figure 4-1 are the tools and algorithms we used to perform the analysis. We use state of the art Motif discovery tool (Kavosh

[12]) and another state of the art tool for clustering based on sparse feature vectors (Bayon [7]).

Motif discovery tool: Kavosh

Kavosh is one of the pioneering tools for discovering k-size Motifs in complex networks for a given k as input. It enumerates all sub-graphs of size k up to isomorphism and works in less memory as compared to any other existing graph theoretic algorithm for Motif discovery[12]. We have to throw away the node type information before we pass on the input to Kavosh. We present the results obtained from running Kavosh on our inputs in section 4.5.

A simple and fast clustering tool: Bayon

Bayon[7] is a simple and fast clustering tool for large-scale data sets. It supports two hard-clustering methods, repeated bisection clustering where you chose the largest cluster and bisect it based on optimality of a cost/distance function and K-means clustering where we iteratively find the new centroids for the data. It utilizes the fact that the feature vectors are sparse and fast hashing techniques can improve the time taken for clustering by a lot. The benefit of repeated bisection clustering over K-means is that we don't have to provide the number of clusters to begin with but instead we have to provide a limit on the value of the cost function for stopping.

Now, Before we use Bayon, we have to transform our DAGs into feature vectors which then onwards can be passed on to Bayon for clustering. We generate feature vectors in a way so as to ensure that the structure and types of nodes is preserved to a greater extent and at the same time the performance (speed) of clustering is not affected much.

Feature Vectors: Generation

This is the most critical part for the success of this phase. We considered many feature vector possibilities so as to find the best representation where "closeness" in multidimensional space corresponds to actual "similarity" between the sub-graphs rooted at particular nodes.

As mentioned earlier in 4.4.1, we divided our input nodes into 8 classes based on similarity in their structure and logic. As a first attempt, we associated with every node a "type bit-vector" of size 8 such that only the bit corresponding to its type is set and the others are all zero. Let's call this feature vector fv_n^0 for a node n. So,

$$fv_{nk}^0 = 1 \iff \texttt{type}(n) = k$$

(assuming all 8 types are represented by the numbers $0 - 7$). Clearly, this feature vector on its own will not help us identify any sub-graphs but it will ensure that the type of this node is the same as any other "similar" node (if the threshold for similarity is set fairly small).

To reach out to more nodes, we can think about appending the fv^0's of a node with fv^0's of its parents. This will help us capture a 2nd level parent neighborhood of each node. But clearly this method will result into an exponential blowup when we look for a 3rd level parent neighborhood and so on. To solve this issue. We introduce

$$fv_n^1 = fv_n^0 || \left(\sum_{p \in pa(n)} fv_p^0 \right)$$

Here $||$ refers to the "append" operation and $\sum$ is the coordinate-wise sum of each vector. This results into $2 * 8 = 16$ dimensions. Note that now each dimension is potentially an integer instead of a bit but even then its very sparse given that usually the types and numbers of parents in our examples are small. This technique can

easily be extended to k-level parent neighborhood where the feature vector for a node is computed by using the k levels of parents above it. As a policy, we do not associate any feature vector fv^k with nodes which do not have any parent in k^{th} level.

We present an example of feature vector calculation for a node till the third level in figure 4-3.

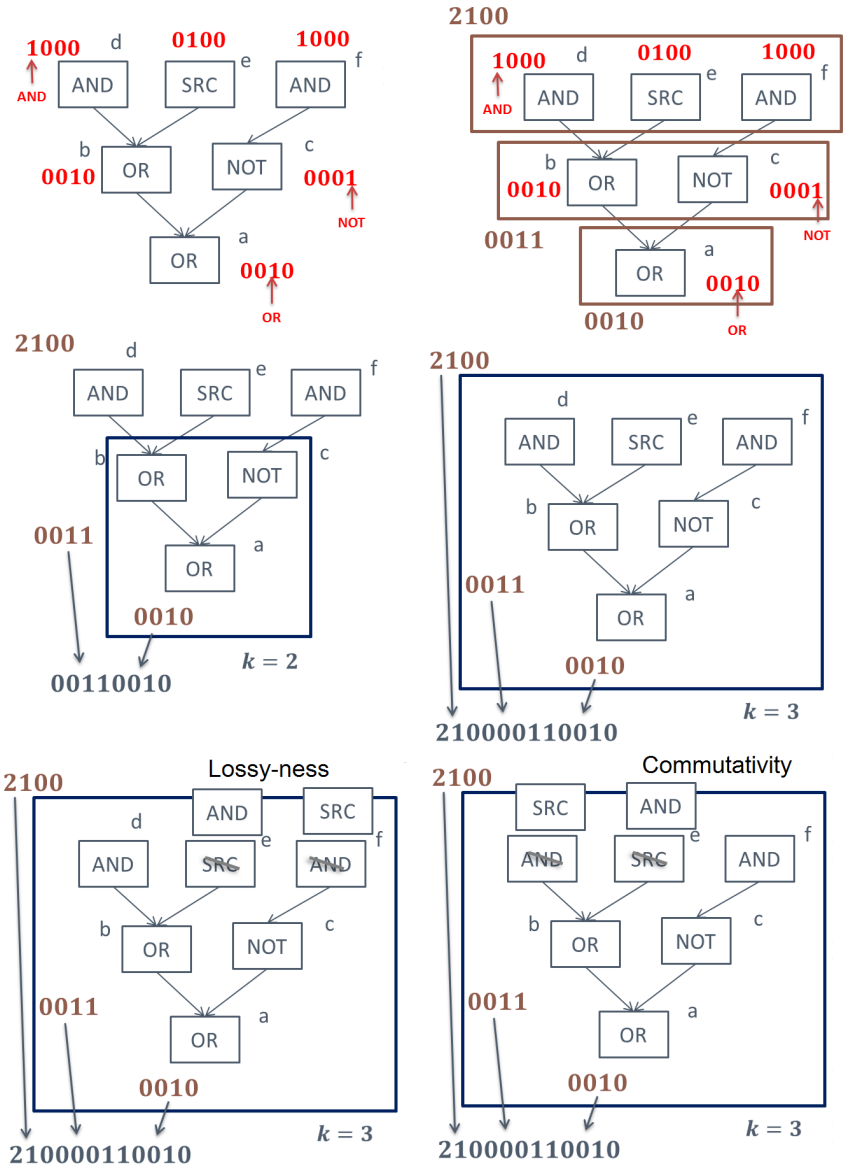


Figure 4-3: Example feature vector calculation

4.4.4 Feature Vectors: Properties

We note some important properties of these feature vector which will justify our choice of the vectors and also potentially explain their limitations.

- These feature vectors lose some information about the structure of the DAG because addition of vectors results into loss of ordering of parents. This is particularly a problem for nodes which are non-commutative.
- But at the same time they help us maintain some properties like commutativity of addition and multiplication (the order of parents may not matter in some cases).
- At every node, we create a window of a particular height rooted at that node. The whole window may not repeat but some smaller sub-structure of it might and these feature vectors help us capture this because the clustering algorithm with find "similar neighborhoods" and the "similarity" may come from a sub-vector of the feature vector as well. When we generate rules for the whole window, it will potentially also generate rules for the recurring part of it as well. This phenomenon is depicted in figure 4-4.

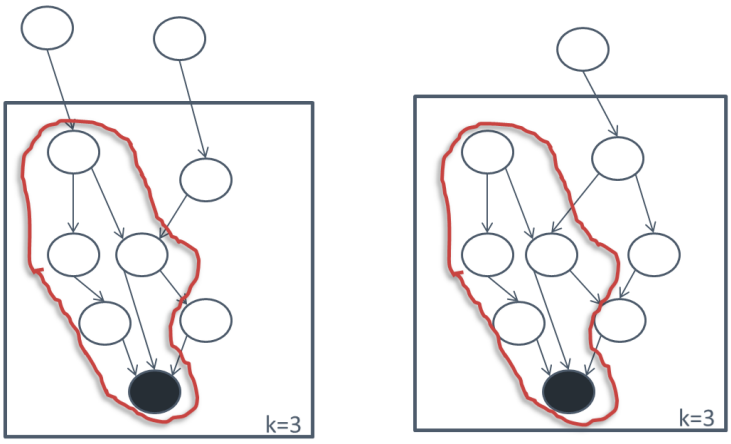


Figure 4-4: Example describing the reason for choosing clustering with such feature vectors

4.5 Some observations

4.5.1 Graph-theoretic motif discovery algorithms: Kovash

While transforming the input to a nascent DAG, we lost all the type information and therefore as expected the results are although much more recurring in the nascent DAG, aren't really much useful for our optimization purposes because the recurrent DAGs can be further classified based on the type of nodes and some of them may be more important than others. All that the results tell us is that the structures which are recurrent are of these particular forms.

Moreover, the tool ran out of memory for the complete DAG join of 13500 nodes. We calculated the Motifs on a smaller subset of around 5 files i.e. 50000 nodes. Even then the time taken for finding Motifs of sizes $4, 5$ and 6 were 2 minutes, 33 minutes and around 2 hours respectively. This processing is just a one time deal and hence such processing times are not completely unacceptable but even then the results are not as useful as the clustering based method (which works for fairly large sizes due to the sparse-ness of the feature vectors). Some of the results for most recurrent and statistically important sub-graphs are presented in figure 4-5. Although, the sub-graphs occur frequently, all the types of the nodes may not be compatible in all of them. For instance, the same parent value being passed to four children (most recurrent Motif with nearly 80% occurrence) can be any value node propagating its output and we cannot really optimize this structure unless we know what is the role played by this value and if the parent node returns a Boolean value or an integer or an array.

4.5.2 fv^k's and the clustering based algorithm: Bayon

We first merged all DAGs obtained from different problems from a single domain. We then computed the feature vectors fv^k's for each eligible node and output it to a

file (Keeping $135,000$ $8 * k$ dimensional integer feature vectors in memory results into a memory overflow on a desktop computer but this file based I/O works well). The feature vectors are provided in the sparse format (readable by Bayon). Each row of the feature vector file looks like:

```
nodeid  f3  2  f7  1  f12  1  f19  3
```

where, each fi is the feature corresponding to the feature vector fv^k and only non-zero features are mentioned. Bayon quickly provides clusters ranked by the "similarity" metric (the default average euclidean distance based cost function). This whole procedure takes around 40 seconds for $k = 3$ which corresponds to including $7 - 12$ nodes in the recurrent sub-structure candidates. The variation of number of "similar" sub-graphs in a cluster are displayed in figure 4-7. We can observe the following from the graphs:

- The number of clusters are higher for larger $k = 3$ and the biggest cluster is not ranked well and hence doesn't really have good "similarity" properties. Even if we can find some rules for some high ranking cluster, it won't be of much significance.
- For smaller $k = 1$, the number of clusters is small and the whole mass is concentrated at the beginning i.e. the best cluster in terms of "similarity" has most of the sub-graphs but this is still not a good result because $k = 1$ means a simple 2-3 node sub-graph which will not help us do much optimizations (figure 4-6).
- $k = 2$ provides many sub-graphs at higher rankings and these graphs have 4-5 nodes which do help us do some optimizations.

We present the best recurrent sub-graphs for $k = 1, 2, 3$ in figure 4-6 for two data-sets QBS and Storyboard.

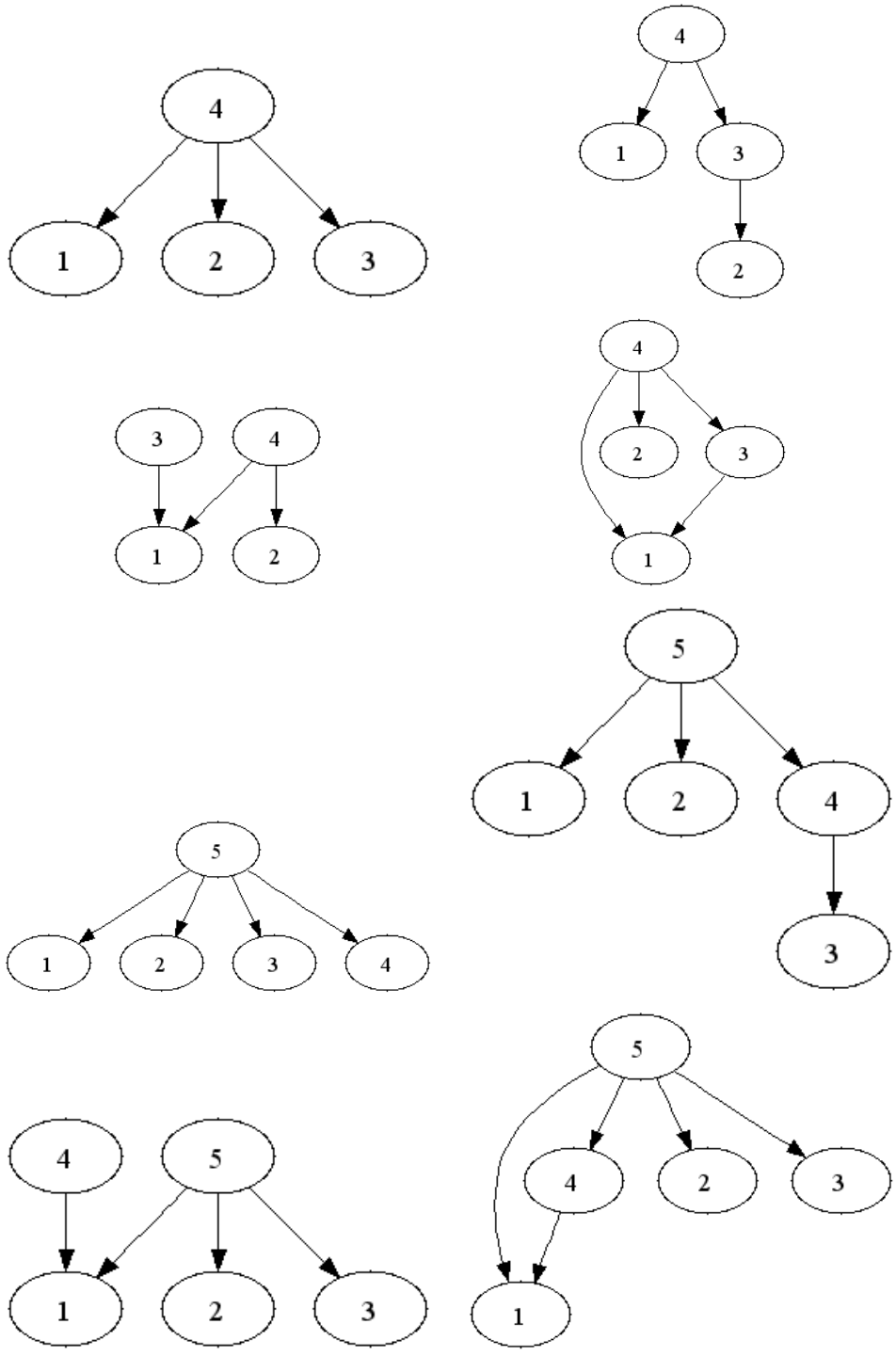


Figure 4-5: Most recurrent Motifs for sizes 4 and 5

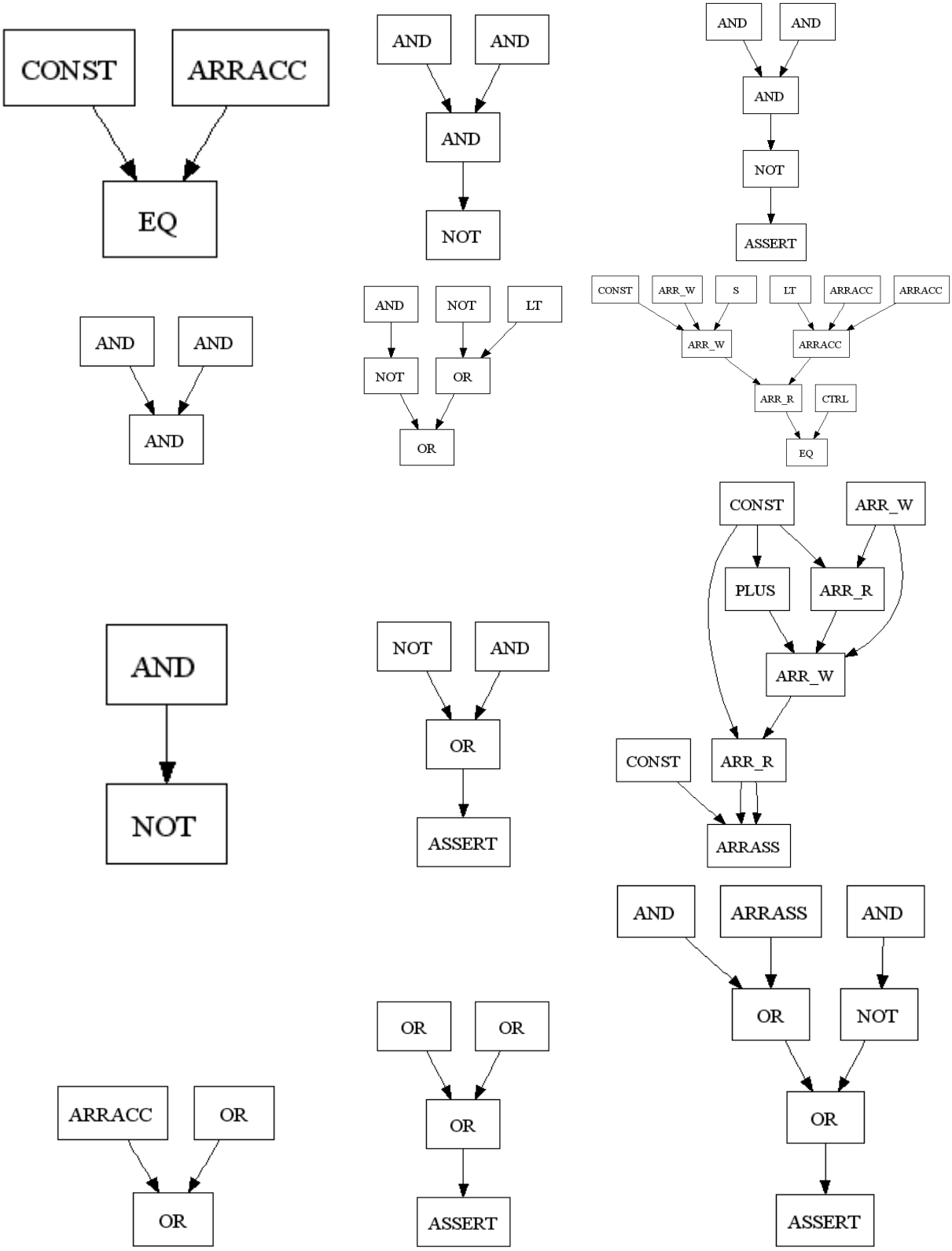


Figure 4-6: Most recurrent sub-structures of the benchmark DAGs. First two rows are for data-set QBS and the second two rows are for Storyboard. First column is for $k = 1$, second for $k = 2$ and the third one for $k = 3$ (depth of the window)

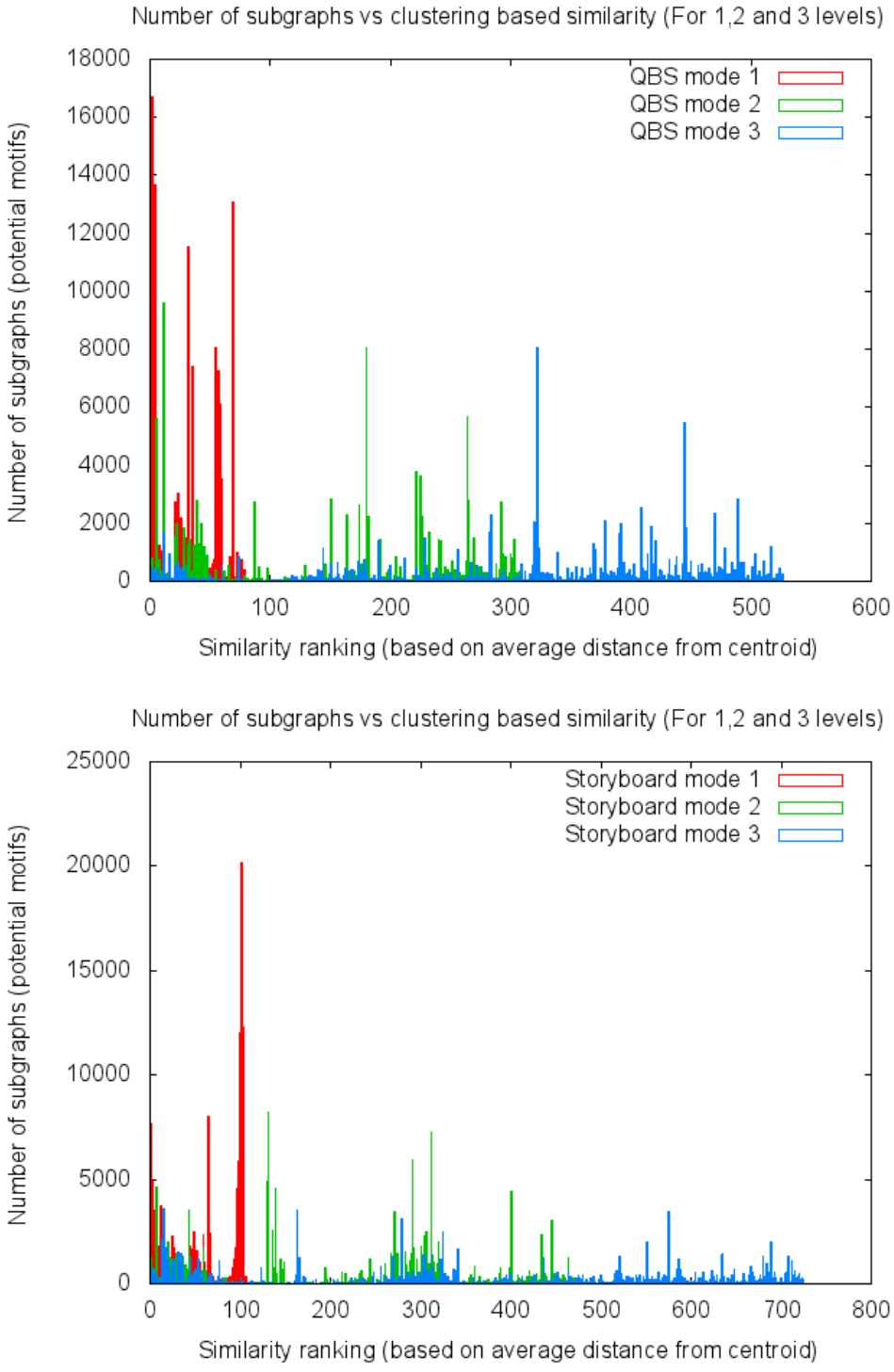


Figure 4-7: Cluster sizes and similarity rankings for k-level parent (fv^k) based clustering

Chapter 5

Synthesis based Rule Generation

After the clustering phase, we have a set of recurrent sub-graphs in the benchmarks from a domain. Now, we would like to find corresponding rewrite rules which can further be used to optimize the problem representation inside **Sketch** .

5.1 Sketch problem formulation

We need to find "correct" rewrite rules (Section 1.3) which have their **LHS** from the set of DAGs obtained after the clustering phase. This correctness constraint will be specified in **Sketch** itself from where we obtained these sub-graphs. Here's how we can formalize this problem:

Problem 2. Given a function **LHS** (x) and templates for **Pred** (x) and **RHS** (x). Find suitable candidates for **Pred** (x) and **RHS** (x) which satisfy the following constraints:

- $\forall x\ if$ (**Pred** (x)) $then\ assert$(**LHS** (x) == **RHS** (x))
- $size$(**RHS**) should be as small as possible for a fixed **Pred** (x)

The problem is to find the **RHS** and **Pred** which enable us to replace **LHS**

(obtained from the clustering phase) whenever the inputs satisfy the predicate **Pred** .

5.2 Sketch representation

We realize the structure mentioned in Problem 2 in **Sketch** using the **generators** and **minimize** features (**??**) of the **Sketch** language. We will use generators to recursively define the templates for **Pred** and **RHS** . We describe both of them in details.

5.2.1 Predicate Pred

For the predicate **Pred** we use a simple Boolean expression generator **bgen** which can be defined as follows:

```
generator bool bgen(generator choice, int bnd){
    assert(bnd > 0);
    if(??){
        //unary operator: NOT
        bool xa = bgen(choice,bnd-1);
        if(??) return !xa;
        //binary operator: OR
        bool xb = bgen(choice,bnd-1);
        return xa && xb;
    }
    else{
        //base case
        a=??;
        b=??;
        return choice(a) ( == | < ) choice(b);
    }
}
```

The generator uses NOT and AND operations with a base case as comparisons of instantiations of another generator (called "choice"). It can produce all possible Boolean formulas with the given base predicates. Note that we perform some symmetry breaking and lazy evaluation of expressions in the actual implementation but the idea of this generator can be conveyed best with this simpler form. In practice, we use only == operation in base case and also remove NOT operation in the upper case because this makes the predicate evaluation easier at run-time when the actual outputs are not available but only their types, estimates and references in the data structure are available.

To use this generator one has to provide another "choice" generator. An example choice generator is given below:

```
... main(int inputs[N]){
    ...
    generator int choice(int x){
        assert(x >=0 && x <= N+1);
        if(x < N){
            return inputs[x];
        }
        else if(x == N) return 0;
        else return 1;
    }
    ...
}
```

The generator *choice* produces a selected value from the inputs array based on its parameter or produces a constant expression 0 or 1. This helps us make the predicates

simple enough so that they can be evaluated during the synthesis process. The bound *bnd* is set based on the problem size. In practice only depth 1 or 2 rules evaluate to *True* during synthesis process.

5.2.2 Recursive function template for RHS

We faced two issues with representing functions within **Sketch** language framework:

1. The generator has to scale well easily for representing a lot of nodes (as we witnessed some **LHS**s of size 15 or higher)

2. It has to be simple enough to be able to get parsed as a DAG with shell scripts.

The first requirement made us consider complicated ways to write recursive generators like the one which generates sets of outputs from each layer and then constructs a layered version of the graph which would have been difficult to parse as a simple DAG although potentially it could have been faster for synthesis purposes. But, since we needed an output dag in the end, we went forward with the generator which simulates the computation of that function using temporary variables as one would do while writing a simple program for it and the DAG naturally gets parsed. Essential elements of the generator for **RHS** is shown here:

```
generator fgen(int inputs[N], int bnd){
    //bnd is the number of nodes allowed
    int output[N+bnd] = inputs;
    //rest of the values beyond N-1 are initialized as 0
    int i=N;
    bool go_on = true;
    repeat(bnd){//unrolls the code inside it bnd times
        if(go_on){
            output[i] = simpleOperation(outputs[0::i]);
            //consider all the values in outputs array
```

```
            //that have been computed already while
            //computing a new value
            i++;
            if(??){
                go_on = false; //opportunity to stop doing operations
    now
            }
        }
    }
    minimize(i); //use the smallest possible i given that rest of the
    program variables are fixed
    return output[i]; //the last computed value
}
```

Here *simpleOperation* is a generator which takes an array and generates an operation as output (representing a node of the DAG, semantics explained in the next section) and we use the **minimize** keyword to find the smallest possible **RHS** DAG. Note that the actual implementation is a bit more complicated because of the fact that we have to consider arrays as both inputs and outputs of the function but the idea behind the sketch is best conveyed by the code shown here.

5.2.3 Representing Sketch back-end/internal semantics in the Sketch language itself

The *simpleOperation* generator needs to take care of the operations and their semantics in **Sketch** language. These must match what actually is assumed in the **Sketch** back-end about these operations on nodes representing values (int/array). At an abstract level its easy to define *simpleOperation*:

```
generator simpleOperation(int inputs[N]){
```

```
    int a = ??;
    if(??) return someUnaryOp(inputs[a]);
    ... //More Unary Operations
    int b = ??;
    if(??) return someBinaryOp(inputs[a],inputs[b]);
    ... //More binary/ternary/ fixed n-ary operations
    int sz = ??;
    int params[sz];
    int i=0;
    repeat(sz){
        params[i] = ??;
        i++;
    }
    return someOtherOpWithArbitraryNumParams(params);
}
```

In the actual implementation we have to take care of the differences between array and integer inputs/outputs and also break some symmetries to make the search faster (like $a < b$ for commutative binary operations). Now, the only thing left to explain is how to interpret the semantics per operations.

We evaluate each operation as **Sketch** would interpret the nodes if it were to "simulate" the computation (**Sketch** does have a simulation phase which does exactly the same). Some operations and data structures used are shown below:

```
int BINOP(int type,int a,int b){
    if(type == PLUS) return (a+b);
    ...
    if(type == LT) return (a < b);
    if(type == AND) {assert ((a==0 || a==1) && (b==0 || b==1)); return
    (a==1 && b==1);}
    ...
```

```
    assert(false);
}

int UNOP(int type,int a){
    if(type == NOT){assert(a==0 || a==1); return 1-a;}
    if(type == NEG){ return -a;}
    assert(false);
}

struct Arr{
    int sz; //size of the array
    int[sz] arr;
    int dflt; //default value
}

int ARR_R(int idx, Arr a){//Array read
    if(idx < a.sz && idx >=0){
        return a.arr[idx];
    }else{
        return a.dflt;
    }
}
```

We have to treat arrays differently from the usual notion, so, we create our own wrapper (a struct) around the usual native array with a default value which is used in case the index is not available in the array, and, it is also the initial value at any index. We may have to extend an array to add a value during array writes and all array operations were built keeping that in mind. We did a deep copy of an array after every write operation so that it matches the semantics of immutable arrays as interpreted by the **Sketch** back-end.

5.3 The complete picture

We already have templates for generating predicates and functions as candidates for **Pred** and **RHS** respectively. Now, we need to put all the pieces together to specify the correctness constraint for rules. We show the idea in the following code (for simplicity we avoid using arrays):

```
//Hard-coded constants based on the output of clustering phase
#define BND_PRED ... //bound on depth of Pred
#define BND_RHS ... //bound on depth of RHS
#define N ... //Number of input variables
#define RHS_MAX_SIZE ... //Same as number of operations/nodes in LHS

int LHS(int inputs[N]){
    //auto-generated after clustering based phase
    int x1 = BINOP(AND,inputs[0],inputs[2]);
    ...
    return x7;
}
//templates for Pred and RHS generation: bgen and fgen
void main(int inputs[N]){
        //choice generator chooses a particular entry from inputs or
   returns 0 or 1
        if(bgen(choice,BND_PRED)){
            assert (LHS(inputs) == fgen(inputs, BND_RHS, f_size));
            //fgen is modified to return its size in a reference
   variable
            assert (f_size < RHS_MAX_SIZE);
            //Assert that f_size is less than RHS_MAX_SIZE
        }
        minimize(f_size); //always find smallest possible RHS
}
```

5.4 Predicate search: Refinement

Now that we have the machinery to generate a "correct" rule, we want to find multiple rules in a structured manner. Since, given a predicate we are already ensuring that the **RHS** is the smallest possible among all possible reductions, now we want to find rules with predicates of different strengths. We realized that there are some (**Pred** ,**RHS**) pairs which are definitely more superior (at least locally) than others. For instance, if we know that $\forall x.\textbf{Pred}_1(x) \implies \textbf{Pred}_2(x)$ and if $size(\textbf{RHS}_1) \geq size(\textbf{RHS}_2)$ then clearly the rule with $(\textbf{Pred}_2, \textbf{RHS}_2)$ is better than the rule with $(\textbf{Pred}_1, \textbf{RHS}_1)$ as $\textbf{Pred}_2$ is true more often than $\textbf{Pred}_1$ and also more effective in terms of size reduction.

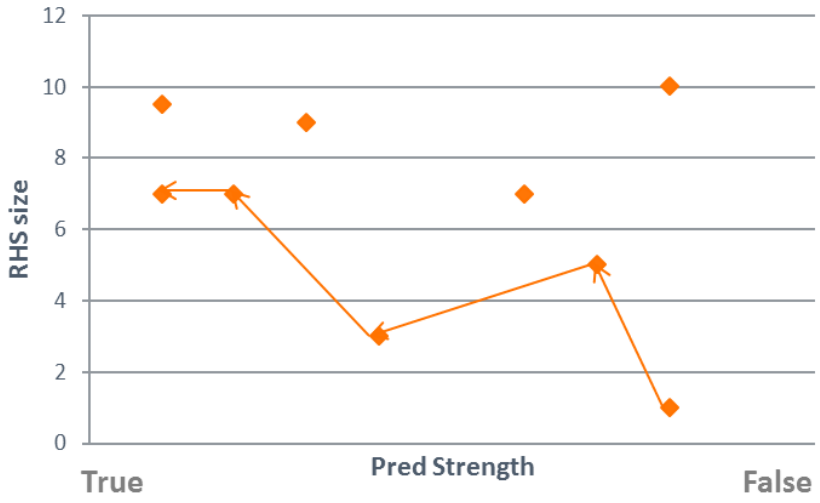


Figure 5-1: Predicate refinement: graphical representation

To avoid capturing such rules, we use a predicate refinement based procedure as shown in table **??**. In the picture shown in figure 5-1, we see that there is an

incomparable boundary of (**Pred** , **RHS**) pairs and those are the ones we want to keep and test for their applicability (arrows show the order of generation of the rules using our algorithm). Note that sometimes we may miss some rules because we started too early in the chain (so its better to start from a rule with considerably larger size so as to make it more probable to start later in the chain).

1. Start with **Pred** $_0$ = **false**. For $i \geq 0$:
2. Find **Pred** $_{i+1}$ such that **Pred** $_i$ $\implies$ **Pred** $_{i+1}$ but **Pred** $_i \neq$ **Pred** $_{i+1}$
3. Find **RHS** $_i$ such that $\forall x$.**Pred** $_i(x)$ $\implies$ (**LHS** (x) == **RHS** $_i(x)$)
4. Repeat steps 2 and 3 as long as its possible to do so.
5. Find another **Pred** which hasn't been seen till now and go back to step 2.

Table 5.1: Predicate refinement: method

5.5 Examples

We conclude this section by presenting rules from two domains which we obtained from the existing benchmark DAGs. These are special in the sense that even after years of optimizations, we weren't able to discover these rules manually. Also, they are applicable with a true predicate: a free optimization available to us. Generating and implementing such complicated rules was not possible earlier, but, due to our automated algorithm, its now possible to efficiently generate and code up these rules.

5.5.1 From Autograder Benchmarks

Consider the rule shown in figure 5-2. It's a simple rule based on Boolean logic. Simple transformations using DeMorgan's laws can give us the proof of correctness as well. The important thing to note here is that since we know that the components of the proof of this rule are very small local transformations, it also means that there are multiple other transformations that can take place for the same **LHS** and we would have no idea which ones to chose while trying to come up with the rule.

The automated method makes it very likely that this particular rule will be used very often in this domain (since the benchmarks have the **LHS** -like patterns in abundance) and it also ensures that this is the smallest possible **RHS** available for transformation under the assumption **Pred** $= true$.

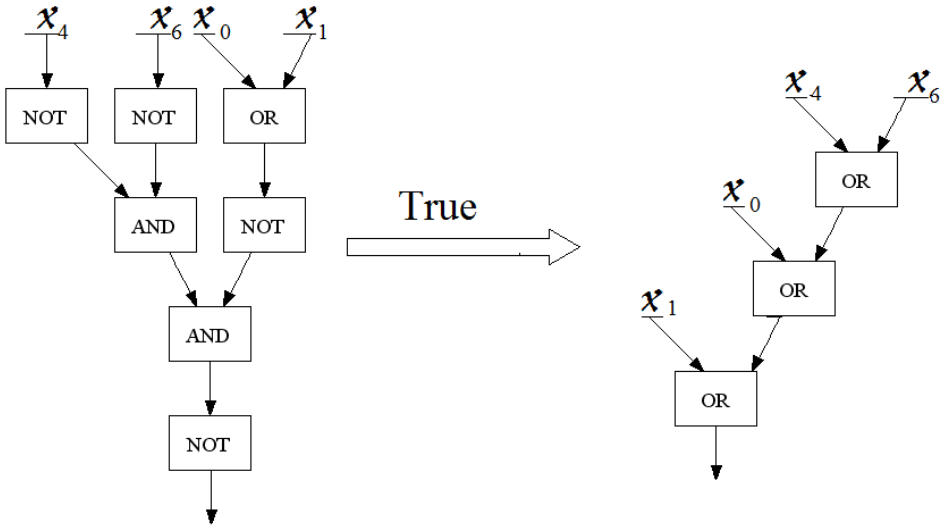


Figure 5-2: Example Rewrite rule from Autograder benchmarks

5.5.2 From Storyboard Benchmarks

Consider the rules shown in figure 5-3. One of the rules (bottom side) works for **Pred** $= True$ which means that it is always applicable. Moreover, the **LHS** pattern is the most common one found in the Storyboard benchmarks. Note that we ran the experiments on DAGs obtained after applying the existing optimizations. So, we were able to find a rule which is heavily applicable (also verified by results) and the existing optimizer couldn't do much about it mainly because its difficult to find such complicated patterns manually and also even when one has seen such patterns, they are very specific to the domain. The pattern matching cost might not justify the rule being present when looking at other domains. Analytically, the rule says that in this pattern there is always a few nodes or operations which are redundant

and independent from the output ($a[x_0 + x_1] = a[x_0]$ statement). Also, we found another rule for the same pattern which reduces the size of the DAG to 1 node from 6 (not counting the inputs) when a simple predicate is true ($x_0 == x_3$). This example brings up some important aspects of our algorithm and its comparison with the manual process, so, we re-iterate them here:

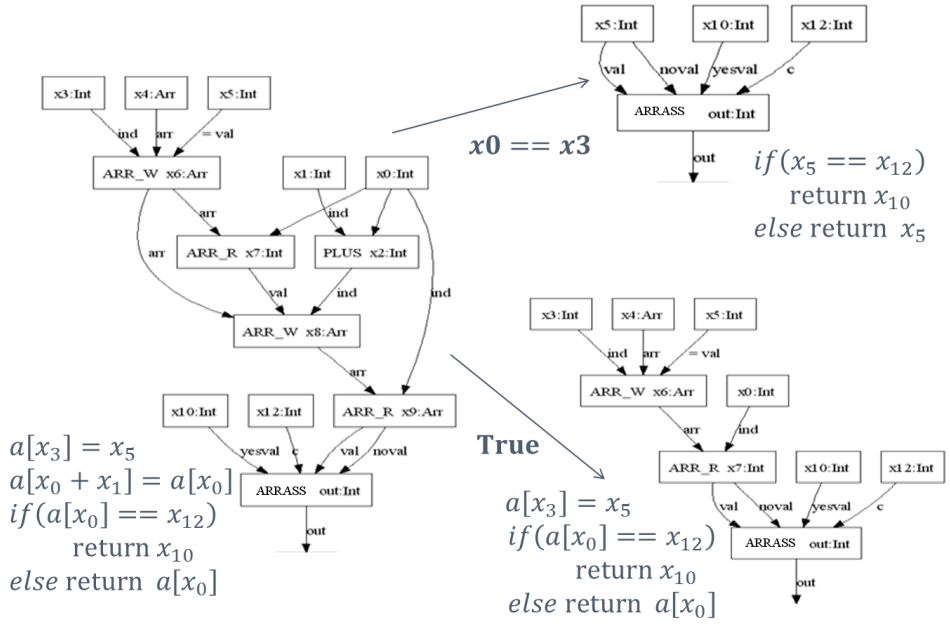


Figure 5-3: Example Rewrite rules from Storyboard benchmarks

- We could find a rule using the automated algorithm which we couldnt find earlier manually
- It is difficult to find huge patterns manually, its difficult to analyze them and even observe that theres an opportunity for optimization
- Even after finding an opportunity, writing replacement code is tedious (high chances of introducing bugs)
- The rules are very domain specific, we don't know if we can add it for other domains since considerable time for matching will be spent

- We obtain multiple rules from one pattern, now they can share the pattern matching code for faster rule application

Chapter 6

Efficient Code Generation for Rewrite Rules' application

We have a set of "potentially useful" rewrite rules for each domain after running the **Sketch** files and parsing the output as triples (**LHS** , **Pred** , **RHS**). We need to generate efficient code for implementing these rules in the **Sketch** back-end. Earlier, the tool developer had to add rules to the tools in a way so as to offer some readability and modularity. Now, there is no need to stick to this constraint. The absence of this readability and modularity constraint offers a possibility of "global" optimization by considering all rewrite rules together. In particular, the most important one being that now we can share the burden of pattern matching among all the rules. This is particularly useful when the **LHS** is the same for multiple rules like the ones shown in figure 5-3. Using this idea and some standard predicate evaluation techniques we can encode the process of rule application much more efficiently than the earlier optimization phase.

To make the process of rule application faster, we first share the pattern matching burden across all **LHS** DAGs from all rules available to us and then secondly, we hard-code all aspects of the rule application including predicate evaluation and replacement

procedure. These two steps ensure that we don't spend a lot of time in pattern matching and also apply rules quickly (avoiding recalculation of temporary variables). We explain the steps in detail:

6.1 Merging LHS DAGs' nodes and obtaining "Matching Map"

The first step in this code generation process is to merge nodes of all the **LHS** DAGs available to us in a way so that nodes with similar predecessors can become indistinguishable and the "decision nodes", the ones where you can be rest assured that the matching with a rule has completed, can be marked easily. We achieve this by going over the DAGs in a topological order one by one and whenever we see a node such that another node of its "signature" has already been visited then we just assign this node the same **id** as the matched node and continue. The "signature" of a node is defined by the **id**s of its parents and the type (operation) of the node itself.So, all source nodes with no parents will get merged together, and, if we see a node of type "DIV" with parent's **id**s being x, y (in order) then its signature is the ordered set (DIV, x, y), we check if there is another node with this signature available in the new merged DAG which is being constructed already (using a hash map for searching). If yes, we just assign this node the same **id** as the node we just found in the map, and, if not, then, we create a new **id** for this node and add its signature to the map pointing to this newly generated **id**. In the end of this pass, we have a merged DAG where each node has a unique signature and the nodes which correspond to output values of the individual DAGs are marked special. The final result from this phase that we need for the next phase is just the hash map which maps "signatures" to **id**s of nodes and also tells which are the output node **id**s (for each **LHS** DAG). The process of constructing this "Matching Map" is explained with the help of an example in figure

6-1 where we merge two maps and in the end to obtain a "signature to **id** map".

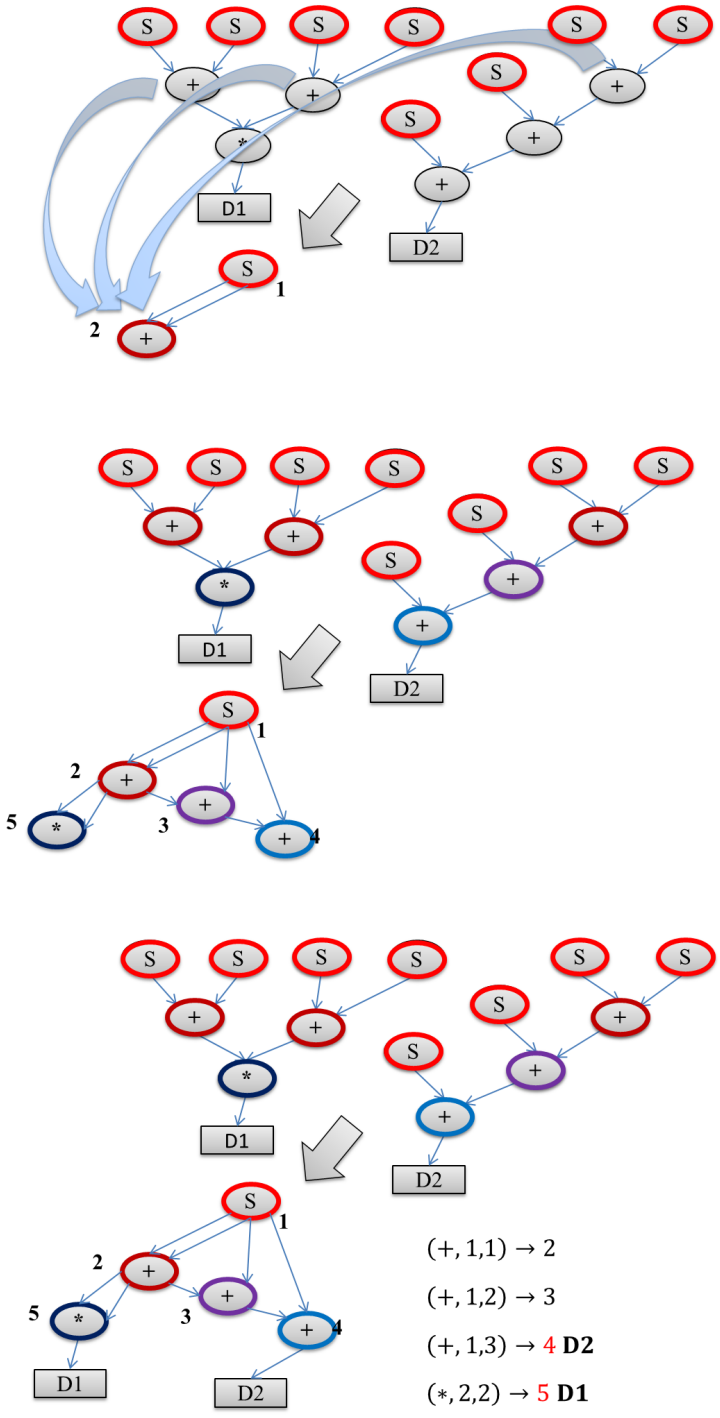


Figure 6-1: Example generation of a Matching map from merging of multiple **LHS** DAGs

6.2 The process of Matching and Replacement

Using the "Matching Map" obtained in the previous section, we can auto-generate efficient matching code and also generate code which will make the replacements. First we show how the matching and replacement procedure can be carried out on a sample problem DAG if we had the "Matching Map" and details of each rule. And then we will show how to generate code which will do the same efficiently.

The matching and replacement procedure begins with assigning a set of integers I_n to each node n in the problem DAG (which we want to optimize) and initialize I_n to be the set of **id**s of all "source" nodes (the nodes which do not have any parents and represent the inputs to the **LHS** DAGs from the rules). Note that we merged all input nodes to the same node in the previous step, so, this instantiation will just make all I_ns a singleton set (This is done for simplification, in practice there can be multiple source nodes based on the type of output they represent like Int, Bool, Array of Int etc).

We traverse through the DAG in a topological order and at the current node n:

- we check if any ordered set of **id**s from parents of n coupled with type of n corresponds to an entry in the "Matching Map" i.e. we find for all possible "signatures" of this node $(type(n), i_1, i_2, ..., i_k)$ such that $i_j \in I_{p_j}$ where node p_j is the j^{th} parent of n (in a fixed order).

- For each such entry: we add the mapped **id** in the "Matching Map" to the set I_n of this node.

- At any stage if we find that this node matches an **id** which is marked as output node of a particular rule's **LHS** DAG, we try to apply the rule:

    1. We first find the "inputs" of the matched **LHS** DAG (note that our algorithm guarantees their existence)

2. Then we check if the predicate **Pred** of the rule is satisfied.

3. If it is not satisfied, we just move on to the next rule whose **LHS** also matched at this node or to the next node if no rule is available.

4. If **Pred** is satisfied, then, we create new nodes and attach them to this base node. Then we run the whole matching algorithm on each new node as well and then proceed in the algorithm to the next node.

An example of this algorithm in practice is shown in figure 6-2.

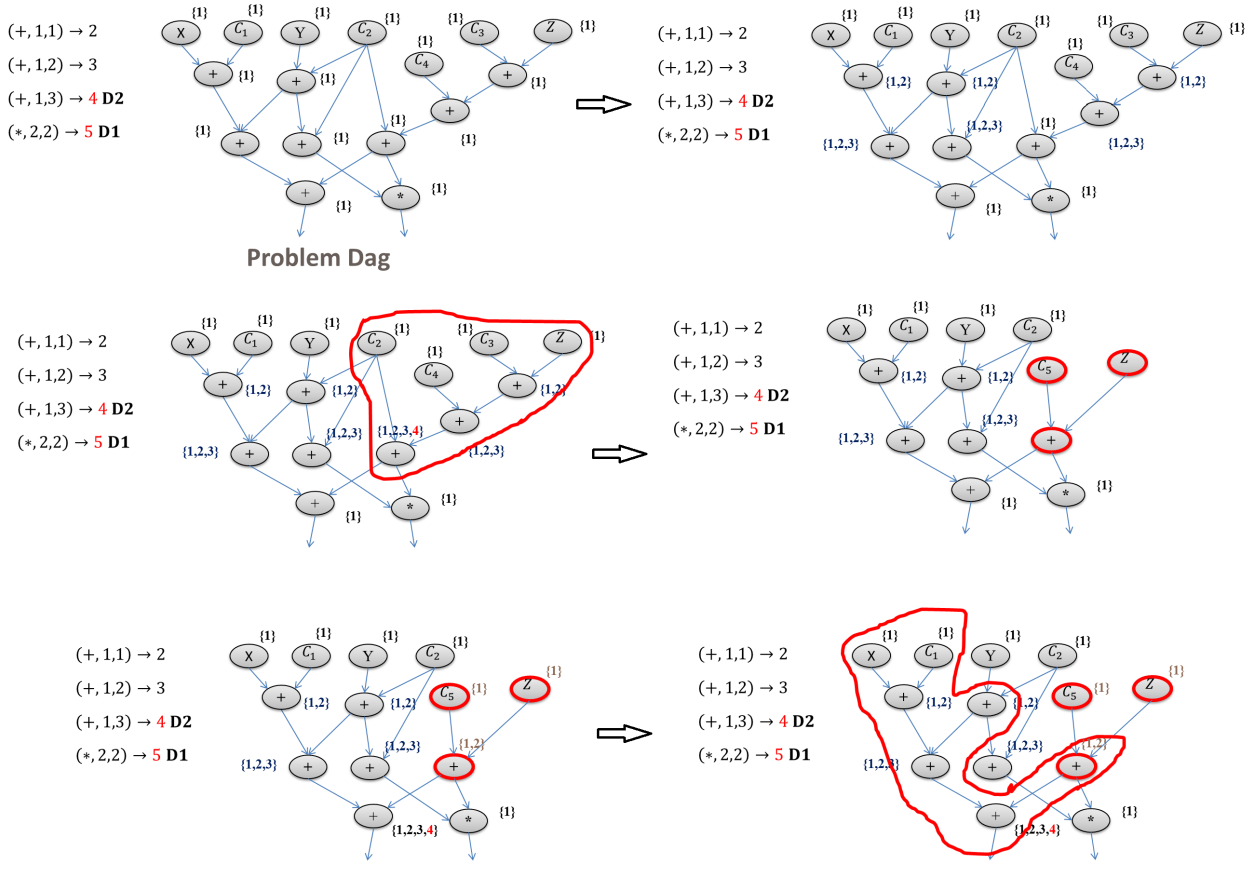


Figure 6-2: Example matching of multiple **LHS** DAGs at the same time, along with application of the rule on the fly

6.3 Predicate Evaluation

Since we do not know the exact values that the node represent, we collect some estimates about them. We wrote a **PredValue** class which compares for "equality", "less than" and other predicates as best as it can (using pointer equality or estimated values) and then combines them for other operations based on a type based set of rules. A PredValue object can be of either of these four types: $Bool(b), Int(i), Pointer(p), Unknown$. All types have an associated value with them of that type except $Unknown$. Equality and other comparisons with the same types (or compatible types like $Bool$ and Int) can be easily evaluated. And, combination operations like "OR" are evaluated differently for each pair of type based on some simple rules like $\_ORBool(\text{true}) = Bool(true)$ where _ represents any value.

6.4 Hard-coding everything

To avoid storing the "Matching Map" in the memory and then looking up the signatures, we just hard-code the matching routine. We assign a set (of potential matched **id**s) to each node. And then we generate the code which will update the set to a new value based on the values of the node's parents (some if...then...else statements with hard-coded values for **id**s). We also generate code for checking if one of the matched **id**s is an output node from one of the rules' **LHS** DAG and then write predicate evaluation and replacement code for that node (which is simply a set of statements creating new nodes, assigning their parents and then disconnecting this node). A sample predicate evaluation and replacement code generated by our tool is shown here:

```
//calculate inputs to LHS DAG
inputsx0 = base->mother->mother;
inputsx1 = base->mother->father;
```

```
inputsx3 = base->father->mother;
bool goahead = false;//evalpred
PredValue t0(inputsx0);
...
t3.binop("EQ",t2,t0);
goahead = t3.value;
//replace if evaluation of the Pred is True
if(goahead){
    map<int,bool_node*> idb;
    idb[0] = inputsx0;
    ...
    idb[3] = new EQ_node();
    idb[3]->mother = idb[0];
    idb[3]->father = idb[2];
    idb[3]->addToParents();
    ...
}
```

This code is highly efficient because not only it takes the benefit of shared pattern matching, it also avoids multiple stacks of function calls (since everything is highly un-modular and hard coded) and creation of useless temporaries (which we usually create only for readability).

Chapter 7

Implementation overview

We built a tool-chain for achieving the automation of the whole process. The major steps of the tool-chain are explained with some details in this chapter. A diagrammatic flow of the tool-chain is shown in figure 7-1.

Benchmark DAG generation (Sketch tool)

Benchmark DAGs are generated from Benchmark **Sketch** specifications using the **Sketch** tool (binary) itself. We use the `--debug-output-dag` flag to get the DAGs in a parsable format.

DAG parsing and Feature-vector generation (C++)

We wrote a simple parser of the DAG output obtained from **Sketch** and generated the feature vectors as per the need of the clustering tool. We marked each node (representing the sub-graph DAG rooted at that node) with its **id** so that we can map the results back to the nodes.

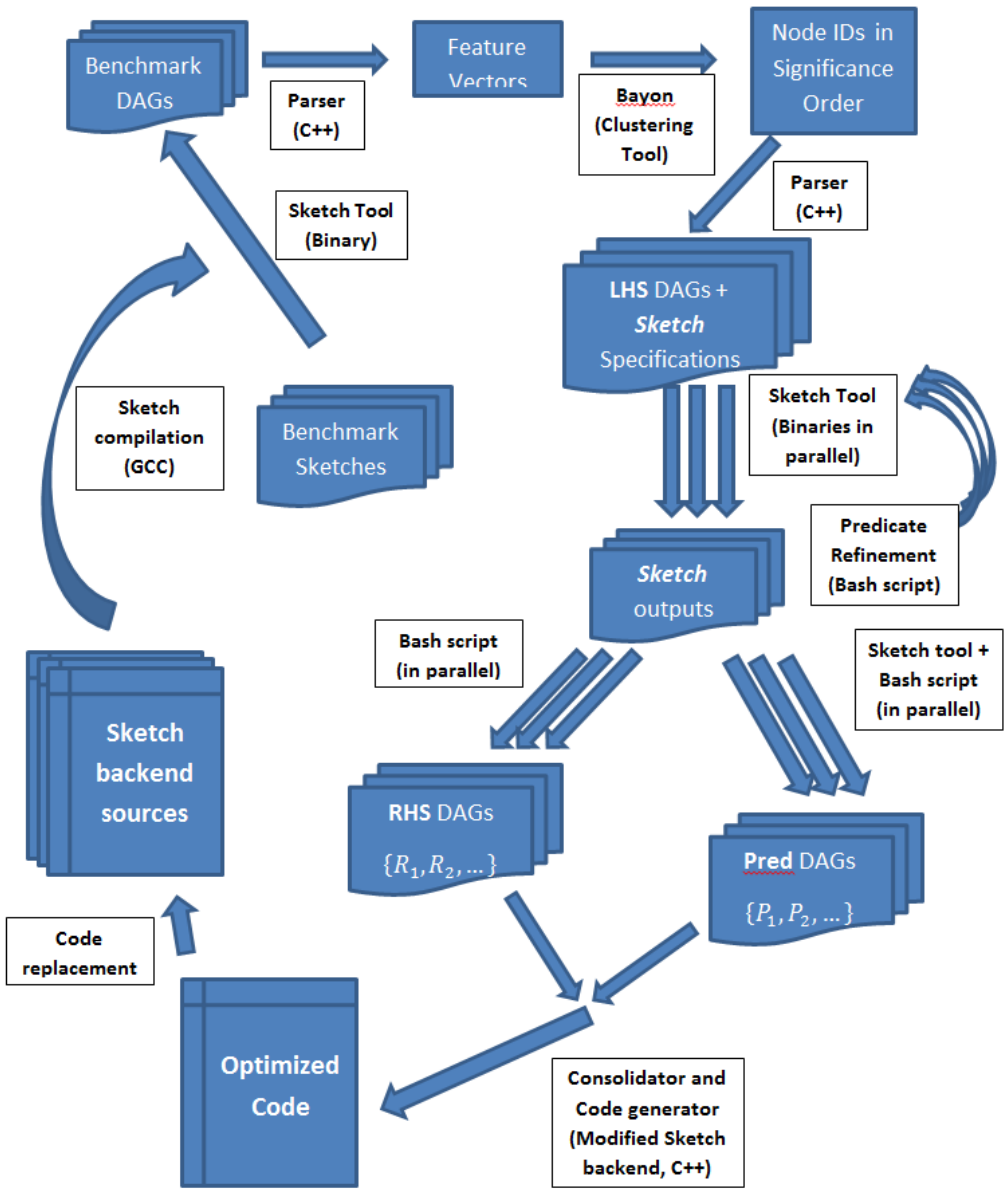


Figure 7-1: Implementation Tool-chain

Clustering (Bayon)

The clustering tool: Bayon [7] takes the feature vector specification and outputs the node **id**s with the corresponding similarity "score" and sizes of clusters. Clustering takes about a minute for the largest data-set.

Sketch specifications' generator (C++)

This tool written in C++ takes the output of clustering tool and in the order of cluster sizes takes a (randomly selected) candidate DAG from each cluster (similarity class) and generates a **Sketch** specification for it (along with the **RHS** and **Pred** generators' library).

Sketch solving and Predicate Refinement (Sketch Tool + Bash script)

The specifications generated by the previous step are solved in parallel using the **Sketch** synthesis tool (binary) and the outputs are stored in different files. Parallel processing is essential for faster solving. It reduced the time taken to generate the outputs from the order of an hour to 10 minutes. We also do predicate refinement and call the **Sketch** tool again a bounded number of times (parameter to the script) with modified **Sketch** specifications which look for a different (refined) predicate.

Extracting the RHS DAGs (Bash script)

Again, in parallel, each **Sketch** output is processed and after some bash scripting using awk,sed,grep etc, we produce the **LHS** DAGs in the same format as supported by **Sketch** `--debug-output-dag` flag, shown in table 4.1.

Recomputing predicates and extracting them (Sketch tool + Bash script)

The predicates are further simplified using **Sketch** (recomputing them and minimizing their sizes because minimizing two costs at the same time doesn't work well in **Sketch** , so we move the **Pred** size minimization to this step). This is also done in parallel for each rule.

Consolidation and Generating optimized code for rule application (Modified Sketch back-end in C++)

We modified the back-end code for **Sketch** to incorporate a new optimization class which can be easily activated/deactivated by a flag. Some part of this class is generated automatically using the set of all rules to be considered. This set can be provided as a sub-set of the original set of rules. We do multiple tests to figure out the best subsets (based on which rules are applied).

Completing the loop, regenerating the benchmark DAGs

Now, we add the optimized code to **Sketch** back-end and again run **Sketch** tool on the original benchmark sketches to generate new DAGs. We can potentially redo the cycle but we stop when the effect of the new rules is not positive.

Chapter 8

Experiments

In these experiments we will show the following:

- validate our hypothesis (on two different domains) that the rewrite rules are domain specific
- Even after years of work going into rewrite rule based optimizations in **Sketch** , we were able to optimize the benchmark DAGs even further

For the latter, we ran our tool-chain on DAGs obtained from **Sketch** after running existing optimizations and rewrite rules (We will call this existing phase "DagOptim" from here on-wards). The two different domains we ran our experiments on, are: (i) Autograder for MOOCs [20] and (ii) Storyboard examples [21]. In both cases, we obtained new rules which are fairly complicated and also reduced the size of the problem even beyond what the state of the art optimizer could do.

8.1 Setup and Methodology

We introduce our analysis as a separate phase of optimization (We will call this new phase "SosOptim" from here onwards: "SOS" is an acronym for "Synthesis of Synthesizer") right at the place in **Sketch** back-end code where the DAGs were extracted

from. We follow the tool-chain shown in figure 7-1 and perform clustering while looking for patterns of depth 3. We order the patterns obtained by their number of occurrences (number of members in the same cluster) and took the top 150 patterns for each domain. These patterns were converted to Sketches and fed to the **Sketch** solver while implementing the predicate refinement technique on top of that. We obtained $0-5$ rules per pattern (most of the patterns were already optimized or required a stronger predicate). Our choice for the grammar was only equalities among input variables and their conjunctions. The rules produced by this phase were transformed into code using our efficient code generator and added to **Sketch** code itself. Finally, the benchmark DAGs were evaluated on the new **Sketch** tool and the difference in the sizes of problem DAG before and after the "SosOptim" process were noted.

8.2 Observations

8.2.1 comparison of "DagOptim + SosOptim" with "DagOptim"

We present our findings in figure 8-1. We obtained $3-6$ rules per domain which reduced the size of all benchmark DAGs and look fairly different from each other. The sizes decreased by 2-10% for each benchmark in both domains. Note that we were able to find these rules even when the existing optimization rules already use many smaller rules which eliminate applicability of composite rules. Which means that the new rules are either "minimal" or are independent of all composites of the existing rules in **Sketch** . Some rules generated by this method are shown in figures 8-2, 5-3 and 5-2.

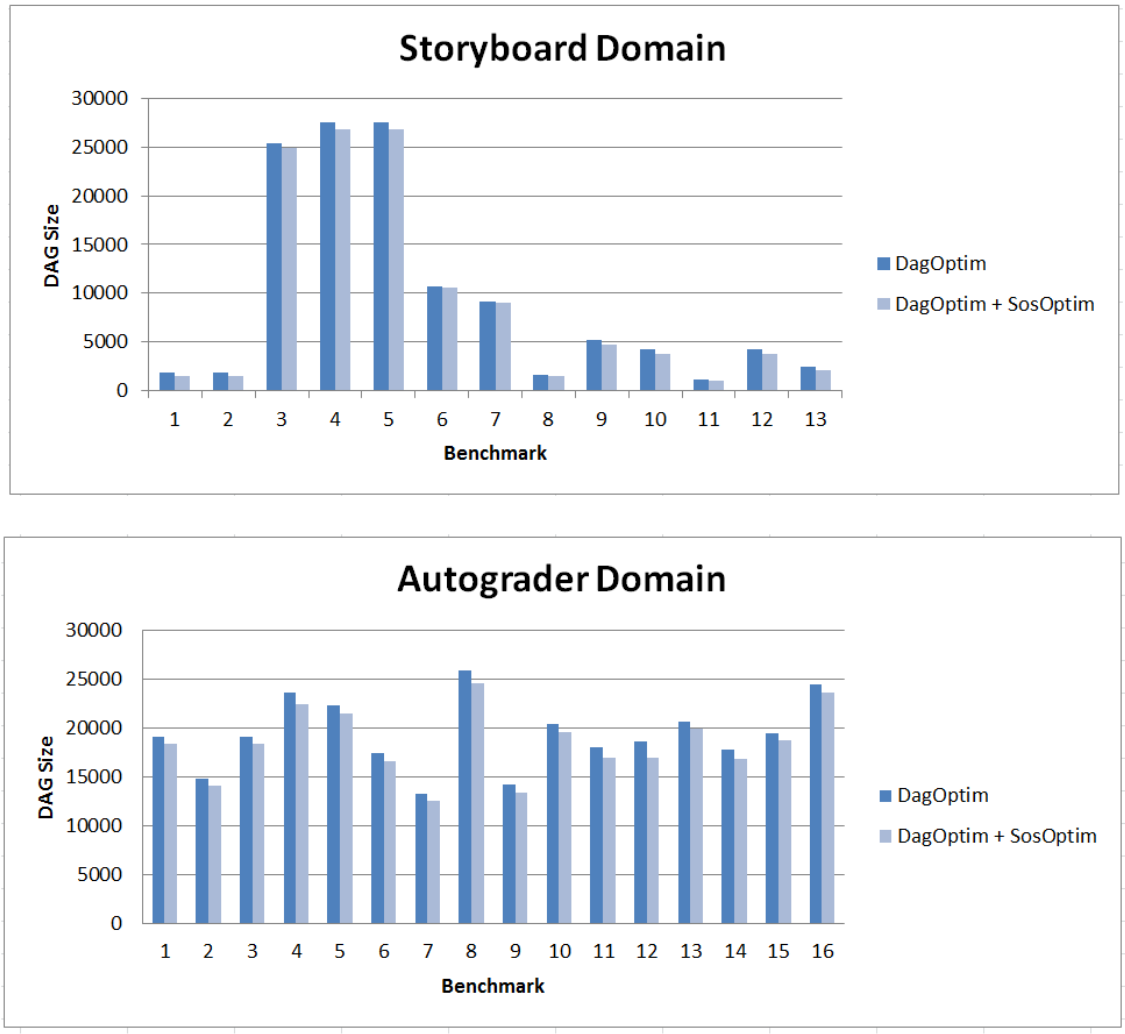
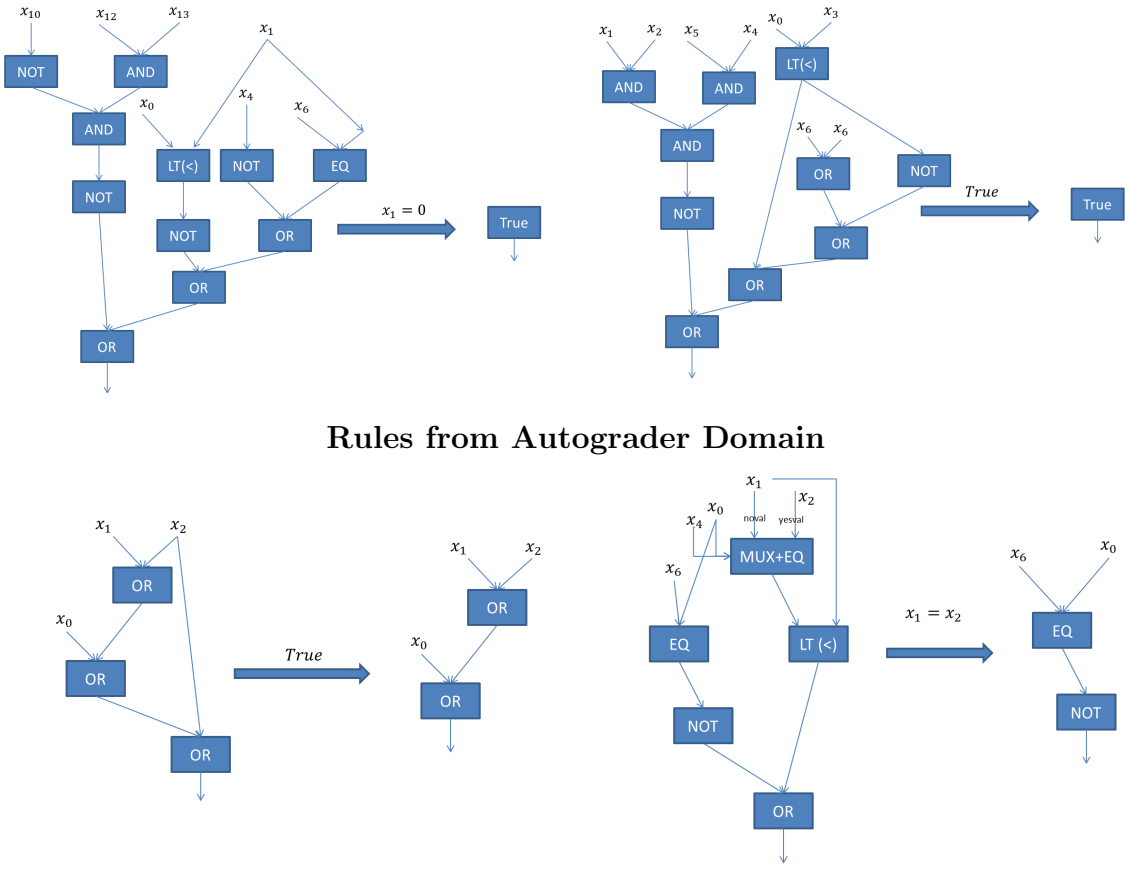


Figure 8-1: Dag sizes of a few benchmarks while running **Sketch** with and without auto-generated optimization code ("SosOptim")

8.2.2 Domain specificity

We already observed that the structure of the rules generated for different domains were exhibiting different patterns (our fundamental hypothesis).We also verified from the clustering phase that the statistically significant patterns in one domain and much different from the other domain. Table 8.1 shows the effect of applying generated rules from one domain on another. As it can be seen, the rules generated by out tool are bigger patterns and complicated enough that they do not apply at all to the other domain.



Rules from Autograder Domain

Rules from Storyboard Domain

Figure 8-2: Rules generated from different domains

Also, we present the data obtained from profiling existing rules from "DagOptim" in figure 8-3. The x-axis shows rewrite rules which were applied significant number of times and the y-axis tells the percentage of the number of times it was used in the respective benchmarks. As one can observe from the graph, many rules which are used for Autograder domain are not used for the Storyboard domain and vice versa. There are many rules that have a huge impact and are applicable specifically to a particular domain.

RULE NO.	AUTOGRADER	STORYBOARD
A1	3521	0
A2	1457	0
A3	832	0
A4	1009	0
S1	0	880
S2	0	476
S3	0	1592
S4	0	984

Table 8.1: Number of applications of rules generated from Autograder domain (Ai's) and rules generated from Storyboard domain (Si's) on benchmarks from both domains

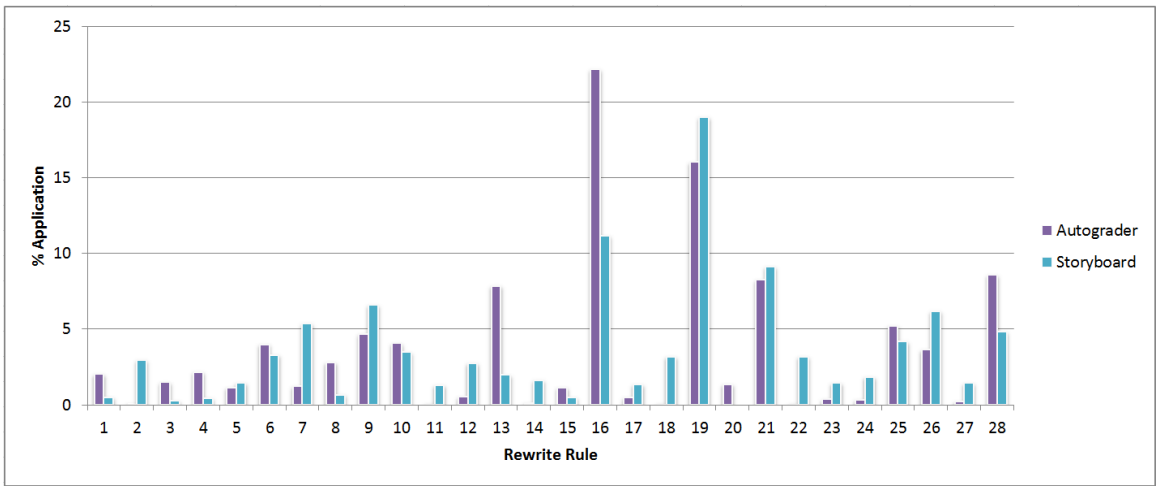


Figure 8-3: Applications of existing rewrite rules from "DagOptim" on different domains

Chapter 9

Conclusion

We conclude by reiterating the contributions made in this thesis and possible future work encompassing the long term vision. We proposed a general automated process of generating "Rewrite Rules" for Solvers and tools employing Solvers (possessing a pre-processing phase). We also tested it for multiple domains in **Sketch** showing the following properties:

1. Relevance and Statistical significance: The rules that are generated are focused towards applicability on the benchmark problems

2. Domain specificity: we verified our hypothesis that the rewrite rules are domain specific and our experiments show that the rules which work for Autograder domain did not work for the Storyboard domain and vice versa

3. Correctness guarantees: **Sketch** applies bounded model checking to its results, so, we can guarantee that at least for integers up to a certain bound the rules are correct. The rules are small enough that it is possible to do full verification on them and we will do that in future.

4. Correct and Efficient implementation of the transformer: We automatically generated code for the transformation phase in the **Sketch** back-end framework

which is bug-free by design and employs efficient pattern matching

This chain of methods will save time for the tool developer and eliminate chances of introducing bugs while writing rewrite rules (since they don't need to write the code or even the rules anymore).

Long term vision

Rewrite rule generation is a critical part of pre-processing step for tools employing solvers. The three steps/building blocks of this step are:

1. Internal representation
2. Rewrite rules
3. Encoding rules

In the long term we would like to automate all three of these i.e. we should be able to suggest most optimal Internal representation, Encoding rules along with rewrite rules for a particular domain. As we have seen in section 1.2, one can and should make domain specific choices for each of them and they can heavily affect the solver's running time. We have achieved automation of Rewrite Rule generation and in the future we want to automate the rest of the chain along with improving this step further.

Bibliography

[1] Nicole E. Baldwin, Rebecca L. Collins, Michael A. Langston, Christopher T. Symons, Michael R. Leuze, and Brynn H. Voy. High performance computational tools for motif discovery. In *IPDPS*. IEEE Computer Society, 2004.

[2] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. ACM.

[3] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[4] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 135–145, New York, NY, USA, 2011. ACM.

[5] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Inferring sql queries using program synthesis. *CoRR*, abs/1208.2013, 2012.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Mizuki Fujisawa. Bayon: a simple and fast clustering tool, 2012. [Online; accessed 9-Dec-2012].

[8] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. pages 383 – 397, 2006.

[9] R. Nigel Horspool. Incremental generation of lr parsers. *Computer languages*, 15:205–233, 1989.

[10] Zhenjiang Hu, editor. *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*. Springer, 2009.

[11] Neil C. Jones and Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms (Computational Molecular Biology)*. The MIT Press, August 2004.

[12] Zahra Kashani, Hayedeh Ahrabian, Elahe Elahi, Abbas Nowzari-Dalini, Elnaz Ansari, Sahar Asadi, Shahin Mohammadi, Falk Schreiber, and Ali Masoudi-Nejad. Kavosh: a new algorithm for finding network motifs. *BMC Bioinformatics*, 10(1):318, 2009.

[13] Mark A. Kon, Yue Fan, Dustin Holloway, and Charles DeLisi. Svmotif: A machine learning motif algorithm. In *Proceedings of the Sixth International Conference on Machine Learning and Applications*, ICMLA '07, pages 573–580, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer, 2004.

[15] Nikolaj Bjorner Leonardo de Moura. Smt: Techniques, hurdles, applications. *SAT/SMT Summer School, MIT*, 2011.

[16] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. An ultrafast scalable many-core motif discovery algorithm for multiple gpus. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 428–434, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[18] Geir Kjetil K. Sandve and Finn Drabløs. A survey of motif discovery methods in an integrated framework. *Biology direct*, 1(1):11+, April 2006.

[19] K. R. Seeja, M. Afshar Alam, and S. K. Jain. Motifminer: A table driven greedy algorithm for dna motif mining. In De-Shuang Huang, Kang-Hyun Jo, Hong-Hee Lee, Hee-Jun Kang, and Vitoantonio Bevilacqua, editors, *ICIC (2)*, volume 5755 of *Lecture Notes in Computer Science*, pages 397–406. Springer, 2009.

[20] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48(6):15–26, June 2013.

[21] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT FSE*, pages 289–299. ACM, 2011.

[22] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[23] Armando Solar-Lezama. The sketching approach to program synthesis. In Hu [10], pages 4–13.

[24] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with kint. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 163–177, Berkeley, CA, USA, 2012. USENIX Association.

[25] Wikipedia. Network motif, 2012. [Online; accessed 9-Dec-2012].

[26] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.

[27] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM.