SIMPLIFIER FOR QUANTIFIER-FREE LINEAR ARITHMETICAL EXPRESSIONS AS A MEANS FOR OPTIMIZING AUTOMATED PROOFS OF PARTIAL PROGRAM EQUIVALENCE

A Thesis
Presented to
The Academic Faculty

by

Maxim Mints

In Partial Fulfillment
of the Requirements for the Degree in
Computer Science in the
College of Computing, Georgia Institute of Technology

Georgia Institute of Technology
December 2018

SIMPLIFIER FOR QUANTIFIER-FREE LINEAR ARITHMETICAL EXPRESSIONS AS A MEANS FOR OPTIMIZING AUTOMATED PROOFS OF PARTIAL PROGRAM EQUIVALENCE

Approved by:

Dr. William R. Harris, Advisor
College of Computing
Georgia Institute of Technology

Date Approved: 2018 Nov 12

I wish to thank David Heath for providing help and guidance to me throughout this project. I would also like to thank Mert Dumenci for his support and helpful discussions.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iv

LIST OF FIGURES AND TABLES vi

SUMMARY vii

CHAPTER

1 INTRODUCTION 1

2 METHODS AND MATERIALS 5

2 RESULTS AND DISCUSSIONS 12

REFERENCES 18

LIST OF FIGURES AND TABLES

Page

Figure 1: One unwinding pass of the algorithm with the simplifier optimization 6

Figure 2: One unwinding pass of the algorithm with the simplifier optimization 7

Figure 3: The DAG representation for a simple expression produced by Z3 8

Figure 4: A simple pair of equivalent programs used as a test target for the expression simplifier 9

Figure 5. A sample unsimplified interpolant formula from one of the Pequod benchmarks 16

Figure 6. A sample simplified interpolant formula 17

Table I. the runtimes of different benchmarks with and without the simplifier used in transforming Z3 output. 13

Table II. the runtimes of the plusRearrange benchmark with and without the simplifier used on CHCs. 14

SUMMARY

The purpose of this study is to introduce performance optimizations and improvements to *Pequod*, an implementation of an algorithm capable of proving or disproving partial equivalence of two computer programs, given their source code or compiled code, without running them. This algorithm can also be re-purposed to solve different fundamental problems, such as proving multithreaded security. Here, partial equivalence of two programs, given matching inputs, means that, if both terminate (i.e. do not loop infinitely), they produce matching outputs. Programs are viewed as sets of procedures (a Java function is an example of a procedure). The following inputs are used: two procedures A and B, one in each program, and some mapping relations correlating the inputs and outputs of A to, respectively, the inputs and outputs of B. The algorithm used in *Pequod* is expected to be far more robust and reliable than any of the currently existing technology for proving partial equivalence, due to being applicable to a far wider range of programs because of the properties of the underlying concept of *product programs*. This technology could find applications in areas such as industry, where it could be used to prove the equivalence of some well-tested implementation with a more optimal replacement, or education, where it could be used to verify correctness of students' solutions to programming problems. With *Pequod*, partial equivalence proofs could extend from being usable in select specific cases to a wide range of possible situations. The optimizations being introduced to, and proposed for *Pequod* mostly revolve around simplifying quantifier-free linear arithmetical expressions produced during the proof process.

CHAPTER 1

INTRODUCTION

A formal definition of program equivalence can be borrowed from [1]: Two functions *(read: programs)* f and f' are said to be partially equivalent (p-equiv for short) if any two terminating executions of f and f' starting from the same inputs emit the same outputs.

Determining partial program equivalence is a task that has been long studied in the field of static program analysis, however, the existing techniques for achieving it have multiple limitations and lack generality [7]. Any program equivalence prover would need to take, as inputs, two programs with some initial procedures, A and B, along with a one-to-one relation mapping the inputs of procedure A to the inputs of procedure B, and a one-to-one relation mapping the output(s) of A to the output(s) of B. It would then need to prove that, for matching inputs, A and B give matching outputs.

A program that could reliably do that would be instrumental in multiple scenarios related to software development, testing, security, and other areas. An example use case in software development would be re-implementing some program or procedure to be more efficient than some existing, verified and tested version. It would be possible to use an equivalence prover to potentially reliably prove that the new, efficient version has the same behavior as the original version (the same inputs produce the same outputs), without requiring potentially extensive dynamic testing, which generally cannot fully guarantee any of the properties being tested. An example of such an application is described in detail in [2]. In computer security, an equivalence prover would also be a powerful tool: it could be used by antivirus software to automate static detection of malicious behavior by comparing procedures in potentially malicious binaries with known malicious procedures from a malware sample. If a match is found, it would mean that the binary being analyzed uses an obfuscated version of the sample.

The techniques for proving partial equivalence are generally based on model-checking, i.e. verifying some condition about a given program. The model-checking approach that will be discussed is based on representing a program as a sequence of states with some conditional transitions between states. Then, a transition between two states can be represented as a Constrained Horn Clause, or CHC, a logical formula consisting of a logical implication.

A CHC is defined to be a predicate logic formula $H(x_s) \Leftarrow \phi \wedge P_1(x_1) \wedge \ldots P_n(x_n)$. In it, the implicant is a logical conjunction of some unknown predicates $P_1 \ldots P_n$ defined over state vector variables $x_1 \ldots x_n$, and some logical constraint ϕ given with respect to some background theory. The consequent $H(x_s)$ is some other, known or unknown, predicate. When applied to model-checking, $P_1 \ldots P_n$ represent the state transition conditions [4].

Given a system of CHCs where at least one CHC has a consequent which is a constrain that needs to be verified by the model checking, we can use a Horn clause solver, such as the one described in [1], to attempt to find possible formulas for the unknown predicates to satisfy the system of clauses. The approach used there is known as Counterexample-Guided Abstraction Refinement (CEGAR), where the constraining condition in the system of CHCs is a negation of the property that needs to be checked, and the predicates are constructed to attempt to satisfy this condition. If this is possible, then there is a way to break the constraining condition, and the model check does not pass. A more specific solver designed specifically for program model checking, DUALITY, is described in [2]. It is also used in this implementation of the equivalence prover.

As of now, the most widely-used approach to proving equivalence uses the notion of sequential composition: processing the two programs under study separately, using model-checking to find if the programs satisfy the condition of having their outputs match. This has its limits: specifically, if at least one of the programs presents an obstacle for which verification becomes an undecidable problem, such as non-linear integer arithmetic (e.g. multiplication), it can be solved only under certain assumptions, as in [5]. This severely restricts the applications of this method, since there are too many frequently occurring cases where obstacles such as non-linear integer arithmetic make the model checker fail or severely degrade its performance. This is due to some specifics of the assumptions made, such as modeling multiplication as a bitwise operation in the same way it would be implemented physically in hardware, which adds dozens of logical variables representing the bits involved in the calculation into the system of CHCs. This can slow down the solver significantly due to the high number of relations between the high number of variables that it must consider.

To solve the issues with sequential composition described above, we demonstrate an approach based on the concept of *product programs*, originally proposed in [6]: "we can define, for any pair of programs c_1 and c_2, a product program c that simulates the execution steps of its constituents". A product program (procedure) of two programs A and B is a program which interwinds the statements of A and B in such a way that, when representing the product program as a system of CHCs, the unknown predicates would be invariants defined over the state vectors (sets of the values of all variables) of A and B. These predicates are known as relational invariants. Then, model checking is used on the product program, using output matching as the constraint condition, to check for equivalence. If the statements of A and B are interwound correctly, all non-linear logic can be "synchronized" around matching states called sync-up points, which allows us to prove the equivalence of the parts of the programs containing non-linear logic without having to verify any conditions for the non-linear logic.

The approach we use to construct product programs is based on representing the sequence of states of a program and their transitions as a Directed Acyclic Graph (DAG). Having such DAGs for A and B, we can find their Cartesian product, as shown in [7]. It can then be made finite by eliminating repeating patterns around sync-up points. This Cartesian product represents all possible ways to interwind program A and program B into the product program, with different “stepping” (sequential state change combination) pairs. Representing the Cartesian product as a system of CHCs with the condition of having the programs’ outputs match allows us to check if any of the stepping combinations is valid, since it would yield a program product which can be used to prove partial equivalence; otherwise, partial equivalence can be disproved. This approach is also to be extended to multiple procedure calls.

For optimization, we simplify the control flow of a program by merging adjacent nodes into Basic Blocks when possible (up until a back-edge, i.e. a cycle, is reached), before “unwinding” it into a DAG with a specific heuristically-determined number of loop unrolling iterations. The approach described is much more robust and powerful than sequential composition, since it works in a far greater number of scenarios and is more efficient and reliable, not needing to use assumptions such as modeling non-linear arithmetic with bitwise operations. As an aside, one of the benefits of the algorithm is that it can easily be extended for statically proving the multi-threaded safety of a single program, assuming the multi-threaded locks are modeled correctly, and if this is the use-case, this specific optimization, though beneficial in other scenarios, should not happen.

CHAPTER 2

METHODS AND MATERIALS

The algorithm has been implemented in the Haskell programming language, using a separate module written in the OCaml programming language, which employs the Sawja library to translate a Java compiled binary into an intermediate format which can then be processed by the Haskell code. Haskell has been chosen as the main implementation language since it allows for an efficient representation of the data types necessary for the algorithm to function, such as various AST (Abstract Syntax Tree) types which are used to model CHCs and other algebraic expressions as trees. It also features a powerful preprocessor, which allows us to define a clean syntax for specifying values of such types. The separate OCaml module is needed since the OCaml-unique Sawja library is currently the most powerful and easy-to-use tool for converting compiled Java binaries that need to be analyzed into a special representation (JBir), which is simple to convert into a control-flow graph; this is all performed on the OCaml side. This graph is then passed to the Haskell side, where it is simplified by merging adjacent nodes, as described above.

For solving systems of CHCs, we use Z3 [5], a theorem prover engine developed at Microsoft. The solution is constructed in multiple passes, as the unwinding of the control-flow graph is constructed, modeling all possible runs through the program separately. On each pass, Z3 constructs a new labeling for each vertex, an inferred logical invariant which is further constrained on every pass. Running DUALITY [4] in Z3 can be computationally expensive, so certain additional optimizations have been made on the Haskell side.

One optimization is based on the concept of inductivity: on a high level, a set of vertex invariants is inductive if it is implied by a set of vertex invariants preceding it in the unwinding. After a single unwinding pass, we will not need to further tighten the constraints on inductive parts of the graph afterwards. However, the inductive invariants and the state transitions, which can be seen as labels on the edges, would still be necessary to constrain other parts of the graph in the subsequent unwinding passes. The inductive portions of the graph are merged into a single state transition edge, labeled with a state transition condition which is a combination all the inductive invariants and conditions, which can be used in the next unwinding pass as part of the system of CHCs passed to Z3.

In this process, equations produced by Z3 are used to form new systems of CHCs, which are then sent to Z3 again. This is shown in Figure 1. Due to Z3 using an inefficient method for simplifying the interpolant expressions it produces as relational invariants, through this process, they can become long enough to significantly hinder computation.

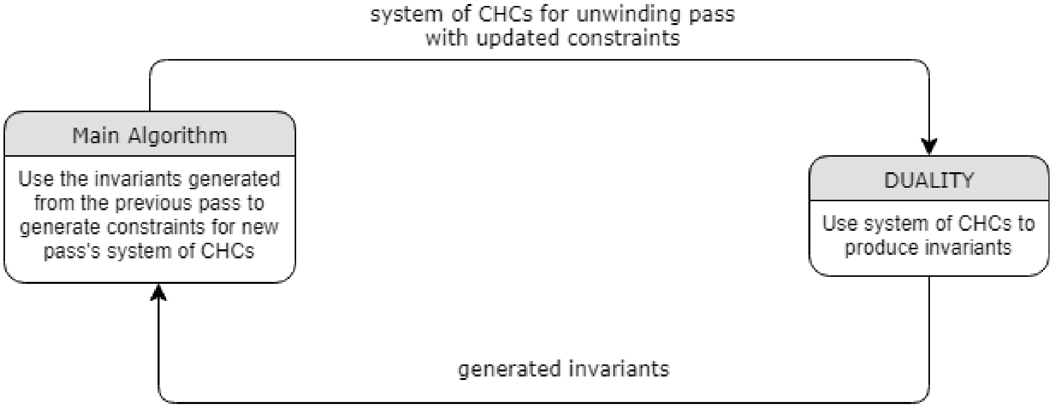

system of CHCs for unwinding pass
with updated constraints
Main Algorithm
Use the invariants generated from the previous pass to generate constraints for new pass's system of CHCs
DUALITY
Use system of CHCs to produce invariants
generated invariants

Figure 1. One unwinding pass of the algorithm

Another optimization technique was introduced to alleviate this issue: an extensible and configurable formula simplifier has been added to the algorithm's interface with the Z3 engine (on the Haskell-side), which, following a given set of rules, transforms typical equations produced by Z3, given as Quantifier-Free Linear Arithmetical (QFLA) Expressions, into much simpler forms which it can later process more efficiently. The updated unwinding pass process is shown in Figure 2. Together, these optimizations are expected to greatly improve performance and further the feasibility of the algorithm.

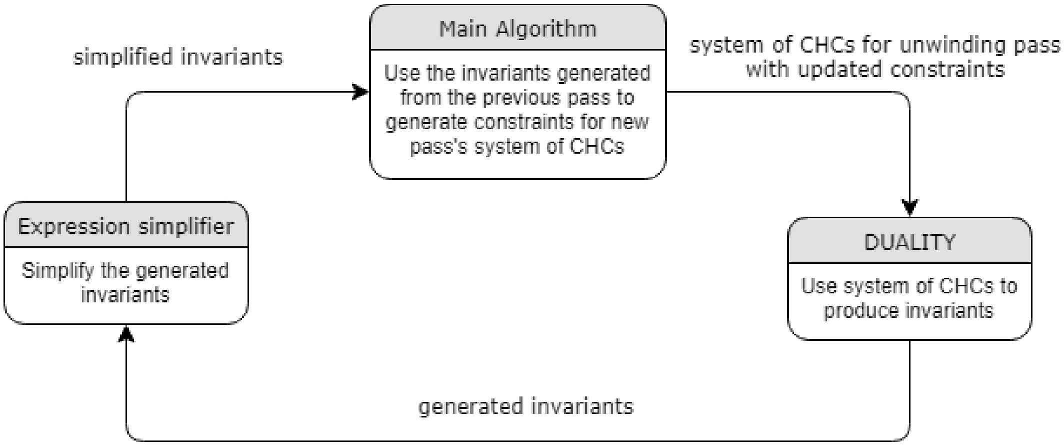

simplified invariants
Main Algorithm
Use the invariants generated from the previous pass to generate constraints for new pass's system of CHCs
system of CHCs for unwinding pass with updated constraints
Expression simplifier
Simplify the generated invariants
DUALITY
Use system of CHCs to produce invariants
generated invariants

Figure 2. One unwinding pass of the algorithm with the simplifier optimization

The expression simplifier works as a rule-rewriting system, by locating and replacing expressions and their subexpressions with simpler forms based on known algebraic identities. A similar technique is used in compiler construction, where complex user-defined expressions are transformed to be more efficient and faster to compute for different values of the variables involved. The main goal for a compiler is reducing the number of costly operations in order to speed up execution time; this simplifier has a similar goal, reducing the number of operations so that Z3 would have less to model, speeding up the execution time of a call to DUALITY on an unwinding pass.

As an example, here is a typical inductive invariant produced by Z3:

$$a+(-b)\geq 0 \land (-a)+b\geq 0$$

Using a set of algebraic identities such as

$$-(x_1+\ldots+x_n)\equiv(-x_1)+\ldots+(-x_n)$$

$$-a\geq -b\equiv a\leq b$$

$$a\geq c \land a\leq c\equiv a=c$$

$$a+(-b)=0\equiv a=b$$

This can trivially be simplified to

$$a=b$$

Which can be modeled and processed by Z3 much more efficiently.

For the purposes of replacement, each expression is converted into a DAG, similar in structure to an AST, except for the fact that nodes which represent equivalent subexpressions are not separate, which makes the algorithm very efficient at detecting patterns in the expression. This is achieved by recursively adding AST nodes to the DAG so that child nodes are added before the parent node, which reduces comparing DAG nodes for equivalence to checking if they have the same operator and same set of child node references. An example of this would be the expression $(!1=0)\lor\left((0\leq !4)\land\left((0\leq((-1*!0)+!2))\land(!1=1)\right)\right)$ (simplified to $(0=!1)\lor\left(((!0\leq !2)\land(0\leq !4))\land(!1=1)\right)$ with a simple carry-over operation). Here, symbols beginning with "!" denote variables. The DAG corresponding to the initial expression is given in Figure 3.

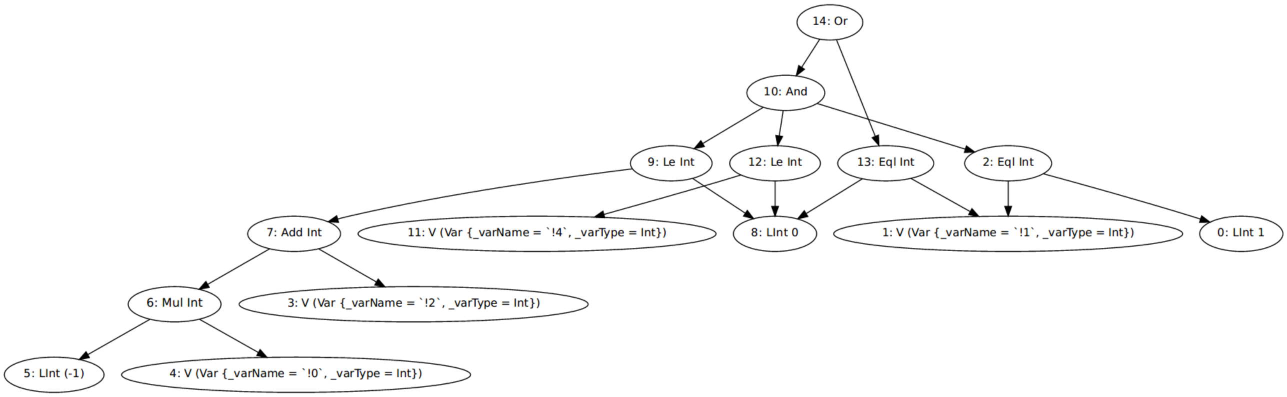

14: Or
10: And
9: Le Int
12: Le Int
13: Eql Int
2: Eql Int
7: Add Int
11: V (Var {_varName = `!4`, _varType = Int})
8: LInt 0
1: V (Var {_varName = `!1`, _varType = Int})
0: LInt 1
6: Mul Int
3: V (Var {_varName = `!2`, _varType = Int})
5: LInt (-1)
4: V (Var {_varName = `!0`, _varType = Int})

Figure 3. The DAG representation for a simple expression produced by Z3:

$$(!1=0) \lor \left((0 \leq !4) \land \left((0 \leq ((-1 * !0) + !2)) \land (!1=1)\right)\right)$$

In such a DAG, commutative and associative operations (addition, multiplication, equality, Boolean and, Boolean or, etc.) with children in the AST having the same operator are converted into DAG nodes by collapsing their operands together, so, in $a+b+c$, there would be one node representing addition, with its children being an unordered collection (a “MultiSet”, a set allowing repetition of elements) containing a , b and c . Subtraction is expanded into addition and multiplication by -1 , e.g. $a-b$ becomes $a+(-1)\cdot b$.

Rule replacement works by first matching rule patterns to the DAG. A rule match pattern is an expression which can contain some variables. The expression is matched to each node in the DAG, going through it with a DFS (Depth-First Search). These variables act as wildcards, matching any subexpression in the DAG. As an example, using the pattern $a+b$ and searching for it in the expression $(x+3)*(x+2)$ would yield 4 possible matches with its two immediate subexpressions, one in which a will be matched to x and b will be matched to 3 , one in which a will be matched to x and b will be matched to 2 , one in which a will be matched to 3 and b will be matched to x and one in which a will be matched to 2 and b

will be matched to x. For associative-commutative operations, a single pattern match variable can be matched to multiple operands, which is handled by creating a faux-node with a subset of the match target's operands and setting it to match some variable in the pattern. A special "remainder" variable can be used for this. They are denoted by names that start with a capital R ("remaining"). Only one R-variable can be used for a single expression node/term. When a pattern with a commutative and associative term having N children, one of which is an R-variable child, is matched to a subexpression X with the same operator and M "collapsed" operands to this operator, where $M > N-1$, then all of the $C(M, N-1)$ possible combinations for matching the pattern's $N-1$ non-R-variable children to some $N-1$ of the M children of X are tried, and, for each of these combinations, a faux-node is created with the same operator as X and the pattern and with X's non-matched $M-(N-1)$ children as its children. This faux-node is then matched to the R-variable. To avoid creating extra patterns with and without the R-variable, for example, for patterns like DeMorgan's law (e.g. so we don't have to create both $M-(N-1)$ and $Ra \vee a \vee \neg b$ as patterns, since searching for both of them in an expression is twice as expensive), each associative-commutative node in a pattern with N children is dynamically assigned an extra R-variable (named Rl*subexpression nesting level*, e.g. Rl1) when being matched to a node with the same operator but with M children, where $M > N$. This, however, presents a major limitation of the mechanism, since it currently doesn't provide a way to match commutative-associative nodes to nodes with the exact same number of children. Also, the naming convention of the Rl-variables does not allow more than one to be used at a given subexpression nesting level (note that the nesting level would be calculated after the children of associative-commutative nodes are collapsed).

When a pattern P is matched to an expression X, some subexpressions of P may get matched to corresponding subexpressions of X in multiple ways. For example, take P to be $a + a \cdot b$ and X to be $y + y \cdot z$, then $a \cdot b$ can be matched to $y \cdot z$ in two valid ways. When backtracking from the recursion, having matched all children/direct child subexpressions of a node in P, we may have multiple match possibilities for each of them, and the match possibilities for this node will be the list of all possible combinations of these subexpressions' match lists. In the above example, these combinations would be: { a matched to y , $a\cdot b$ matched to $y\cdot z$ with a matched to y and b matched to z } and { a matched to y , $a\cdot b$ matched to $y\cdot z$ with a matched to z and b matched to y }. Note how in the second combination a is matched to different nodes in the different constituent matches. This means that this combination of matches can't form a match, and it is therefore eliminated. As a result, there is only one match – the pattern $a+a\cdot b$ is matched to $y+y\cdot z$, with a matched to y and b matched to z .

After some patterns get matched to a target expression, each match location can be replaced with some replacement expression. If the variables of the match pattern occur in the replacement rule, the subexpressions originally matched to these variables will be substituted into the replacement. For example, with an example like $(x+3)\cdot(y+2)$, if we apply the match pattern $a+b$ with a replacement rule $a+a\cdot b$, the result could be $(x+x\cdot 3)\cdot(y+y\cdot 2)$ (this is not the only possible match for this match pattern, but, if there are more than one possible match for an expression by the time all backtracking is done and replacement is to be applied, an arbitrary single match possibility is chosen). Any Rl-variable used in the substitution that was not dynamically created as part of a match will be removed from the expression, e.g. its parent will be created with one less child. It is not allowed to have an Rl-variable as the only thing in the pattern replacement, since it's not guaranteed for it to be present.

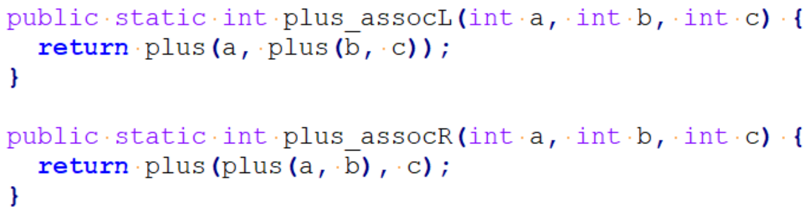
The implementation of the simplifier, including some documentation, can be seen at https://github.com/Mints97/rewrite-simplifier/blob/master/src/Formula/Simplifier.hs. It is a robust and highly configurable system, which is used to provide it with a match pattern decision tree crafted specifically to simplify formulas of the type that frequently occur within *Pequod* in a highly efficient manner.

CHAPTER 3

RESULTS AND DISCUSSIONS

The performance of this algorithm applied to simplifying invariants has been tested on multiple simple pairs of programs used as benchmarks. Unfortunately, at this time, the expression simplifier has not been shown to yield a significant performance benefit in most cases: constructing the DAG for each expression, matching various rule replacement patterns on it and performing the actual rewriting takes approximately as much time as is gained with each individual simplification by passing the simplified QFLA to Z3.

The most probable cause for this would be the fact that, on each iteration, the system of CHCs is modified constrained with the expressions produced by Z3 by eliminating segments of the CHC system with the help of Z3's built-in *entails* operation, which attempts to prove that one formula implies another, and is much less computationally expensive than many other Z3 operations. Despite this, the expression simplifier has already been shown to be performing efficient rule rewrites: for the interpolants produced by processing a simple pair of programs shown in Figure 4, the simplifier achieved a reduction of length (in characters) of approximately *31.54%* on average (when counting only the interpolants it saw fit to process). An example of an interpolant from this pair of programs is given above and in Figure 3.



```
public static int plus_assocL(int a, int b, int c) {
  return plus(a, plus(b, c));
}

public static int plus_assocR(int a, int b, int c) {
  return plus(plus(a, b), c);
}
```

Figure 4. A simple pair of equivalent programs used as a test target for the expression simplifier.

However, in several cases, a small but measurable consistent benefit can be gained from directly applying the simplification algorithm to formulas produced from Z3. This may be caused by the fact that they're generated less frequently than the interpolants which were subject to simplification in the proposed algorithm. Originally, one of Z3's two built-in simplifiers were used on these formulas, however, each of them proved to be detrimental to performance, taking more time to simplify the expression than was gained from the simplification, so they were eventually removed. For several existing *Pequod* benchmarks (made from pairs of programs), the original version of the code applying no simplifications to the formulas produced by Z3 was compared to the version applying the new rule-rewriting simplifier introduced in this paper to each formula produced by Z3. Each benchmark is run twice, averaging out the timings. Testing was performed on an Intel Core m processor (number of CPUs is irrelevant since the application is single-threaded, not yielding itself well to parallelization due to inconsistencies in data allocations boggling down the ghc multithreaded garbage collector). The data is shown in Table I.

Benchmrk Name	First run with simplifier (sec.)	Second run with simplifier (sec.)	Simplifier: average (sec.)	First run without simplifier (sec.)	Second run without simplifier (sec.)	Without simplifier: average (sec.)
beqNatTrans	18.042	18.266	18.154	18.384	18.311	18.348
evenbS	34.166	34.110	34.138	34.388	34.391	34.390
multAssoc	39.582	39.741	39.662	40.219	39.938	40.079
multDistL	34.928	35.032	34.980	34.734	36.330	35.532

Table I. the runtimes of different benchmarks with and without the simplifier used in transforming Z3 output.

An additional modification that was tried involved the simplification of CHC formulas directly before passing them into Z3. In most cases, that had a negative effect on performance, except for some benchmarks, for example – the plusRearrange benchmark, where performance drastically improved. However, in this situation, a similar improvement could also be achieved by applying one of Z3's built-in simplifiers in a similar manner, since, apparently, for that specific benchmark, consistency of structure in ASTs being fed into Z3 had a tremendous impact on performance. In Table II, runtime data is given for all 3 cases: simplification of CHC formulas with algorithm described in this paper, simplification with built-in Z3 algorithm, and no simplification.

Algorithm used on CHCs	Z3 builtin simplifier used on CHCs	No simplifier used on CHCs
2m 56.000s	3m 3.9380s	8m 50.985s

Table II. the runtimes of the plusRearrange benchmark with and without the simplifier used on CHCs.

In this situation, the performance advantages offered by the simplifier are far greater than the ones shown in Table I. However, specific reasons for why this happens with this specific benchmark should be studied in order to use this effectively. A feasible direction to take could be creating a separate rule-replacement decision tree for simplifying CHCs that will come as inputs to Z3's DUALITY.

An alternative option which could yield a performance boost with the simplifier is to phase out Z3's DUALITY in favor or a system which is better suited to the types of CHCs produced when analyzing program equivalence. Such a system, Shara, is currently being developed [9]. It is highly probable that using the simplifier on intermediate results during the interpolation steps of the solver, which still delegates its core computations to Z3, could yield increased performance.

Additionally, in the future, extra processing could be done on the formulas to further simplify them through the structural minimization technique proposed by Lampert [8]. Compared to the current pattern-matching rule-rewriting system, which handles LIA (Linear Integer Arithmetic), this algorithm represents the QFLA expression as a First-Order Logic (FOL) formula, ignoring the structure of the linear terms.

Even without making use of Lamport's algorithm, extensive simplifications could be extremely useful when debugging an application such as Pequod. Figures 5 and 6 show an example of how the simplification algorithm performs on extremely large interpolant formulas that can't be processed by Z3's builtin simplifiers, yielding an approximate 50% decrease in size in the general case.

Despite its drawbacks, this expression simplification method has been shown to be quite promising, with great potential for optimizing implementations of model-checking algorithms relying on QFLIA formulas, as well as helping with debugging such implementations. Also, it is generally a useful tool for working with such formulas, allowing to search and modify them freely, in a manner similar to regular expressions. More research should be done into adapting this system to Z3's DUALITY and other similar algorithms, constructing rule-replacement decision trees corresponding to different types of formulas needing simplification, and finding ways to reliably combine multiple simplification approaches which have varying effects on the performance of different benchmarks.

```
((a1 = a2) && ((a6 = a7) && ((a0 = 0) && ((a5 = 0) && (a3 = 0))))) || (((not (a4 = 1)) && ((not (a5 = 1)) && ((a0 <= 1) && ((a0 >= 1) && ((a3 = 0) && ((a6 = a7) && (not (a5 =
2)))))))) || (((a4 = 1) && ((not (a3 = 0)) && ((1 <= a0) && ((a3 <= 0) && ((a0 <= 1) && (not (a5 = 1))))))) || (((not (a0 = a4)) && ((a0 = 0) && ((not (a5 = 0)) && ((1 <= a4) &&
((not (a3 = 0)) && (not (a4 = 1))))))) || (((not (a0 = a4)) && ((a3 = 0) && ((a4 = 0) && ((not (a0 = a5)) && (a5 = 1))))) || (((a5 = 1) && ((not (a0 = 1)) && ((a3 = -1) && (not (a0 =
-1))))) || (((not (a0 = 0)) && ((not (a0 = a4)) && ((a5 = 1) && ((a4 = 0) && ((1 <= ((-1 * a1) + a2)) && ((-1 <= (a1 + (-1 * a2))) && (((not (((-1 * a1) + a6) <= -1)) || ((not ((a1 + (-
1 * a6)) <= 1)) || (a2 = a7))) && ((not (a3 = -1)) && ((not (a3 = 0)) && (not (a0 >= 2))))))))))) || (((not (a0 = 0)) && ((not (a0 = a4)) && ((a4 = 0) && ((not (a5 = 1)) && ((a5 = 0)
&& (((a0 = a2) || (not ((-2 + a0) = a1))) && (a3 = 0))))))) || (((a1 = a2) && ((not (a0 = a5)) && ((not (a3 = 0)) && ((a0 = 0) && (a4 = 0))))) || (((not (a4 = 0)) && ((not (a5 = 1)) &&
((not (a5 = 0)) && ((2 <= (a7 + (-1 * a6))) && ((not (a0 = -1)) && ((a3 <= -1) && ((-1 <= a3) && ((0 <= a4) && (not (a4 = 1)))))))))) || (((a0 = 0) && ((not (a3 = 0)) && ((a3 <=
2) && ((2 <= a3) && ((a4 = 0) && (a1 = a2)))))) || (((a1 = a2) && ((a0 = 0) && ((not (a5 = 0)) && ((not (a4 = 1)) && ((-2 + a7) = a6))))) || (((a4 = 0) && ((a0 <= 1) && ((1 <= a0)
&& ((not (a5 = 0)) && (((a2 = 1) || ((not (a1 <= 0)) || (not (a1 >= 0)))) && (((not (((-1 * a1) + a6) <= -2)) || ((a2 = a7) || (not ((a1 + (-1 * a6)) <= 2)))) && ((a5 <= 2) && (not (a5 <=
1))))))))) || (((not (a5 = 0)) && ((not (a3 = 0)) && ((a3 = 1) && ((a4 = 0) && ((not (a0 = 1)) && ((2 <= (a2 + (-1 * a1))) && ((-2 <= ((-1 * a2) + a1)) && (a0 <= 2)))))))) || (((not
(a5 = 1)) && ((a3 = 0) && ((a4 = 0) && ((-2 <= (a6 + (-1 * a7))) && ((not (a0 = 0)) && ((((2 + a6) = a2) || (not (a1 = a6))) && (a0 = 2))))))) || (((a4 = 0) && ((a0 = 0) && ((3 <=
(a7 + (-1 * a6))) && (not (a3 = 0))))) || (((not (a3 = 0)) && ((not (a0 = a5)) && (((not (a0 <= 2)) || (not (a0 >= 2))) && ((not (a0 = 1)) && ((a3 = 1) && ((not (a4 = 1)) && ((not (a4
= 2)) && ((a4 = 0) && (a5 = 0))))))))) || (((not (a5 = 0)) && ((a4 = 0) && ((not (a5 = 1)) && ((not (a3 = 1)) && (((not (a1 = a7)) || (not (a1 = a6))) && ((-2 <= ((-1 * a2) + a1)) &&
((2 <= (a2 + (-1 * a1))) && (((a2 = a7) || ((not ((a2 + (-1 * a6)) <= 4)) || (not (((-1 * a2) + a6) <= -4)))) && ((not (a3 = 0)) && ((a0 = 2) && (a3 = 2))))))))))) || (((not (a0 = 0)) &&
((-1 <= ((-1 * a7) + a6)) && ((a3 <= 1) && ((a4 = 0) && ((((-1 * a6) + (a7 + a1)) = a2) && ((1 <= ((-1 * a6) + a7)) && (a5 = 0))))))) || (((not (a5 = 1)) && ((a3 <= 1) && ((a0 = 0)
&& ((a3 = 1) && ((a4 = 0) && ((not (a6 = a7)) && (a1 = a2))))))) || (((a4 = 0) && (((a2 = a7) || (not (a1 = a6))) && ((not (a5 = 1)) && (((a1 <= -2) || (not (a2 = 0))) && ((not (a0
<= 1)) && ((not (a0 = 0)) && ((a5 = 0) && (((a1 <= -1) || (not (a2 = 1))) && (-2 <= (a6 + (-1 * a7)))))))))))) || (((3 <= a3) && ((not (a0 = 0)) && ((a0 <= 1) && ((a0 >= 1) && ((a4
= 0) && ((not (a5 = 1)) && (a5 = 0))))))) || (((3 <= a3) && ((not (a0 = a5)) && (((not (a2 = 0)) || (not (a1 = 0))) && ((a4 = 0) && (((not (a1 = a6)) || (not (a1 = a7))) && ((a5 = 0)
&& (not (a0 = a3)))))))) || (((a5 = 0) && ((not (a0 = 0)) && (((not (a1 = 0)) || (not (a2 = 0))) && (((a1 = 0) || (not (a1 = a4))) && ((not (a3 = 1)) && ((a4 = 0) && ((not (a0 = a3))
&& ((not (a3 = 0)) && (a0 = 1))))))))) || (((a0 = 0) && ((not (a5 = 1)) && ((a5 = 0) && ((a3 <= -2) && ((not (a4 = 0)) && ((not (a4 = 1)) && (a4 = 2))))))) || (((a5 = 1) && ((not
(a4 = 0)) && ((1 <= ((-1 * a1) + a2)) && ((3 <= a4) && ((a0 <= 1) && (not (a3 = -1))))))) || (((not (a5 = 1)) && ((not (a4 = 0)) && ((a0 = 1) && ((a4 = 2) && ((2 <= a3) && (not
(a5 = 0))))))) || (((not (a0 = 0)) && (((not (a1 >= 0)) || (not (a2 = 0))) && ((a3 <= 1) && (((not (a0 <= 1)) || (not (a0 >= 1))) && ((-1 <= ((-1 * a7) + a6)) && ((a5 = 0) && (a4 =
0))))))) || (((not (a4 = 0)) && ((not (a0 = 0)) && ((not (a5 = 0)) && ((a3 <= 1) && (((not (a0 <= 1)) || (not (a0 >= 1))) && ((not (a5 <= 1)) && (((a7 + (-1 * a6)) >= 2) && ((2 <= ((-
1 * a1) + a2)) && ((not (((-1 * a0) + a4) <= -2)) && (a3 <= -2)))))))))) || (((a3 <= 1) && ((not (a5 = 0)) && ((not (a4 = 0)) && ((not (a5 = 1)) && (((a7 + (-1 * a6)) >= 2) && ((a0
= 1) && (not (a5 = 2)))))))) || (((a3 <= 1) && ((not (a5 = 0)) && ((not (a4 = 0)) && ((not (a0 <= 1)) && ((6 <= ((-1 * a6) + a7)) && (((a7 + (-1 * a6)) >= 2) && ((not (a0 = 0)) &&
((not (a5 <= 1)) && ((not ((a4 + (-1 * a0)) <= -2)) && (((a1 + (-1 * a2)) <= -2) && ((not (a0 = 3)) && (a5 = 3)))))))))))) || (((not (a5 = 0)) && ((not (a0 = 0)) && ((a3 <= 1) &&
(((a7 + (-1 * a6)) >= 2) && ((a5 = 2) && ((a0 <= 2) && ((a4 = 2) && (((a2 + (-1 * a1)) >= 2) && ((not (a0 <= 1)) && ((-5 <= ((-1 * a7) + a6)) && (5 <= ((-1 * a6) + a7)))))))))))) ||
(((not (a4 = 0)) && ((not (a4 <= 1)) && ((not ((a4 + (-1 * a0)) <= -1)) && ((not ((a4 + (-1 * a0)) <= -2)) && ((not (a0 = 0)) && (((a7 + (-1 * a6)) >= 2) && ((a3 <= 1) && (((a1 +
(-1 * a2)) <= -2) && ((2 <= a0) && ((a3 <= 0) && ((not (a5 = 0)) && ((not (a5 <= 1)) && ((1 <= (a4 + (-1 * a0))) && (a0 <= 2))))))))))))) || (((not (a0 = 2)) && ((a5 = 2) &&
((not (a4 = 1)) && ((0 <= a0) && ((a3 <= -1) && (a0 <= 1)))))) || (((not (a4 = 1)) && ((not (a0 = a3)) && (((a2 = a7) || (not (a1 = a6))) && ((not (a5 = 2)) && ((a5 = 0) && ((not
(a6 = a7)) && ((a0 <= 2) && ((not (a3 = 2)) && ((not (a3 = a4)) && ((not (a0 = 0)) && (not (a4 = 0)))))))))))) || (((not (a4 = 1)) && ((not (a5 = 2)) && ((not (a0 = 0)) && ((a5 = 0)
&& ((3 <= a4) && ((1 <= (a4 + (-1 * a0))) && (a3 = 0))))))) || (((a4 = 1) && ((not (a5 = 0)) && ((a5 = 1) && ((-1 <= ((-1 * a2) + a1)) && ((1 <= (a2 + (-1 * a1))) && ((a0 <= 1)
&& ((not (a0 = 0)) && ((a3 = 0) && (2 <= (a7 + (-1 * a6)))))))))))) || (((not (a5 = 2)) && ((not (a0 = a4)) && ((a4 = 1) && ((not (a0 = 0)) && ((a5 = 0) && (a3 = 0)))))) || (((not (a5
= 1)) && ((a4 = 1) && ((not (a0 = 0)) && (((a2 = a7) || (not (a1 = a6))) && ((a5 = 0) && ((not (a0 = 2)) && ((not (a6 = a7)) && (((not (a2 = 1)) || (not (a1 = 1))) && (not (a0 =
a3)))))))))) || (((not (a4 = 0)) && ((a5 = 1) && (((a2 + (-1 * a1)) >= 1) && ((not (a4 <= 1)) && ((not (a3 = 0)) && ((a0 <= 1) && ((1 <= a0) && (a3 <= -1)))))))) || (((a0 <= 1) &&
((not (a4 = 1)) && ((not (a0 = 0)) && ((a3 = 0) && ((not (a5 = 0)) && ((not (a0 = a4)) && ((a5 = 1) && ((not (a4 = 0)) && ((-2 <= ((-1 * a7) + (a6 + (a2 + (-1 * a1))))) && (2 <=
(a7 + ((-1 * a6) + ((-1 * a2) + a1)))))))))))))) || (((a5 = 1) && ((1 <= ((-1 * a1) + a2)) && ((not (a0 = 0)) && ((not (a4 <= 1)) && ((not (a3 = -1)) && ((a0 <= 1) && (not (a3 =
0)))))))) || (((a5 = 1) && (((not (a0 <= 1)) || (not (a0 >= 1))) && ((not (a4 = 0)) && ((a0 >= 1) && ((a0 <= 2) && (((2 + a1) = a2) && (not (a3 = 0)))))))) || (((not (a5 = 1)) && ((a4
= 1) && ((a0 = 2) && ((a3 = 1) && ((2 <= (a2 + (-1 * a1))) && ((-2 <= ((-1 * a2) + a1)) && (((not ((a2 + (-1 * a6)) <= 4)) || ((a2 = a7) || (not (((-1 * a2) + a6) <= -4)))) && (not (a5
= 0))))))))) || (((a3 <= 0) && ((not (a0 = 0)) && ((a4 = 1) && ((a5 = 1) && ((0 <= a3) && (not (a0 = 1))))))) || (((a4 = 1) && ((not (a0 = 0)) && ((a5 = 1) && (((not (a2 = 1)) || (not
(a1 = 1))) && ((a0 = 1) && ((not (a3 = 1)) && ((not (a3 = -1)) && (not (a3 = 0))))))))) || (((not (a5 = 1)) && ((not (a0 = 0)) && ((a3 = 0) && ((a4 = 1) && (((not (a0 <= 1)) || (not
(a0 >= 1))) && ((-2 <= (a1 + (-1 * a2))) && ((2 <= ((-1 * a1) + a2)) && (a5 = 2)))))))) || (((not (a5 = 1)) && ((a4 = 1) && (((3 <= a2) || (not (a1 = 1))) && ((a0 <= 2) && ((not (a0
= 0)) && ((not (a3 = 1)) && ((not (a5 = 0)) && (not (a3 = 0))))))))) || (((not (a5 = 0)) && ((a4 = 1) && ((a5 = 1) && (((not (a2 = 1)) || (not (a1 = 1))) && ((a0 = 1) && ((((2 + a2) =
a7) || (not (a1 = a6))) && ((1 <= (a2 + (-1 * a1))) && (-1 <= (a1 + (-1 * a2)))))))))) || (((not (a5 = 1)) && ((not (a5 = 0)) && ((a0 <= 1) && ((not (a0 = a3)) && ((not (a5 = 2)) &&
((3 <= a5) && ((not (a4 = 0)) && (not (a3 = 0))))))))) || (((not (a5 = 1)) && ((not (a0 = 2)) && ((a5 = 2) && (((not (a2 = 1)) || (not (a1 = 1))) && ((a0 = 1) && ((not (a4 = 2)) &&
((not (a4 = 0)) && (((a1 <= -1) || (not (a2 = 0))) && (not (a3 = 0)))))))))) || (((not (a0 = 0)) && ((a4 = 1) && ((a5 >= 3) && ((not (a5 <= 1)) && (((a1 + (-1 * a2)) <= -2) && ((not
(a0 >= 1)) || ((a4 = 1) && ((not (a5 <= 1)) && (((a1 + (-1 * a2)) <= -2) && ((a0 <= 2) && ((not (a0 = 0)) && (a0 >= 2)))))))))))) || (((a0 <= 1) && ((not (a0 = 0)) && ((not (a5 = 2))
&& ((not (a5 = 1)) && ((a5 = 0) && ((not (a6 = a7)) && ((not (a4 = 2)) && ((not (a0 = a3)) && ((a4 = 1) && ((a0 = 1) && (2 <= a3))))))))))) || (((not (a4 = 1)) && ((not (a3 = 1))
&& ((a5 = 0) && ((not (a6 = a7)) && ((not (a0 = 0)) && ((not (a3 = 0)) && (((not (a2 = 0)) || (not (a1 = a4))) && ((a0 = a4) && (2 <= ((-1 * a1) + a2))))))))) || (((not (a0 >= 2))
&& ((not (a0 = 0)) && ((a5 = 2) && ((a0 <= 1) && ((a0 >= 1) && ((a4 = 2) && ((1 + a1) = a2))))))) || (((2 <= ((-1 * a1) + a2)) && ((not (a3 <= 1)) && ((a5 = 0) && ((not (a0 =
0)) && ((not (a0 >= 3)) && ((a4 = 1) && ((a3 <= 2) && ((not (a0 <= 1)) || (not (a0 >= 1)))))))))) || (((a5 = 0) && ((not (a4 = 0)) && ((not (a3 <= -1)) && ((not (a3 >= 2)) && ((not
(a0 = 0)) && ((not (a4 = 1)) && (((not (a0 <= 1)) || (not (a0 >= 1))) && ((not ((1 + a4) = a0)) && (1 <= a3)))))))) || (((not (a0 = 0)) && ((not (a4 = 2)) && ((2 <= a0) && ((a4 = 1)
&& ((a5 = 0) && ((a3 <= 1) && ((a2 = a7) || ((not (((-1 * a1) + a6) <= 0)) || (not ((a1 + (-1 * a6)) <= 0)))))))))) || (((not (a0 = 0)) && ((not (a6 = a7)) && ((a5 = 0) && ((1 <= ((-1 *
a1) + a2)) && ((1 <= a0) && ((1 <= (a2 + (-1 * a1))) && ((-1 <= ((-1 * a2) + a1)) && ((not (a3 = 0)) && ((a0 = 1) && ((a3 = 1) && (3 <= a4))))))))))) || (((not (a4 = 0)) && ((not
(a4 = 2)) && ((1 <= a0) && ((a6 = a7) && ((a5 = 0) && (a3 <= -2)))))) || (((not (a4 = 1)) && ((1 <= a0) && ((a6 = a7) && ((a5 = 0) && ((a3 <= -1) && ((not (a4 = 0)) && (not
(a0 = 2)))))))) || (((a6 = a7) && ((not (a4 = 1)) && ((a0 = 0) && ((a5 = 0) && ((a3 <= -3) && (not (a4 = 0))))))) || (((not (a0 = 0)) && ((a5 = 0) && ((0 <= a0) && ((a3 <= -1) &&
((not (a4 = 0)) && (not (a3 = a4))))))) || (((3 <= ((-1 * a6) + a7)) && ((not (a4 <= 1)) && ((a0 <= 2) && ((not (a4 = 0)) && ((not (a0 = 0)) && ((a0 >= 2) && (((a2 + (-1 * a1)) >=
1) && (((a2 + (-1 * a1)) >= 2) && ((not (a0 <= 1)) && (((a2 + (-1 * a1)) <= 2) && (a5 = 1)))))))))))) || (((a0 <= 2) && ((not (a0 = 1)) && ((a5 = 1) && ((not (a0 = 0)) && ((a3 = 0)
&& (3 <= a4)))))) || (((not (a5 = 1)) && ((not (a5 = 2)) && ((a0 = 2) && ((not (a4 = 0)) && ((not (a4 = 1)) && ((not (a5 = 0)) && (3 <= a5))))))) || (((not (a4 = 1)) && ((not (a4 =
0)) && ((not (a0 = 0)) && ((a5 = 0) && ((a0 <= 1) && ((3 <= (a7 + (-1 * a6))) && ((2 <= a3) && ((3 <= ((-1 * a6) + a7)) && ((not (a3 <= -1)) && ((a3 >= 2) && ((a4 = 2) &&
(a0 >= 1)))))))))))) || (((1 <= (a2 + (-1 * a1))) && ((not (a0 = 0)) && ((a0 <= 1) && ((a0 >= 1) && ((a5 = 0) && ((a4 = 1) && ((not (a0 >= 2)) && (((a1 + (-1 * a2)) >= -1) &&
(((a7 + (-1 * a6)) >= 1) && ((1 <= ((-1 * a1) + a2)) && (not (a3 = 0)))))))))))) || (((1 <= a4) && ((a0 <= 1) && ((not (a0 = 0)) && ((a5 = 0) && ((a0 >= 1) && ((a3 = 1) && ((not
((a7 + (-1 * a6)) <= 0)) && (((a2 + (-1 * a1)) >= 1) && (((a2 + (-1 * a1)) <= 1) && (not (a4 = 1)))))))))) || (((a4 = 0) && ((not (a5 = 1)) && ((not (a0 = a3)) && ((a0 = 1) && ((not
(a5 = 0)) && (not (a3 = 0))))))) || (((a4 = 0) && ((not (a0 = 1)) && (((not (a1 = 0)) || (not (a2 = 0))) && ((not (a5 = 2)) && ((not (a0 = 0)) && ((not (a5 = 0)) && ((not (a5 = 1)) &&
(not (a0 = a5))))))))) || (((not (a5 = 0)) && ((a3 = 1) && ((not (a0 = 1)) && ((a5 = 1) && ((not (a0 = 0)) && (a4 = 0)))))) || (((a3 = 1) && ((a4 = 0) && (((not (a2 = 0)) || (not (a1 =
0))) && (((a2 = 1) || (not (a1 = 0))) && ((not (a5 = 0)) && ((a5 = 3) && (not (a0 = 3)))))))) || (((a4 = 0) && (((not (a0 <= 1)) || (not (a0 >= 1))) && ((2 <= ((-1 * a1) + a2)) && ((not
(a0 = 0)) && ((not (a0 >= 3)) && ((a3 = 3) && ((not (a0 <= 1)) && (a0 = a5)))))))) || (((not (a5 = 0)) && ((not (a5 = 2)) && ((a4 = 0) && ((a5 = 1) && ((a0 <= 1) && ((3 <= a3)
&& ((1 <= (a2 + (-1 * a1))) && (-1 <= ((-1 * a2) + a1))))))))) || (((1 <= ((-1 * a1) + a2)) && ((not (a0 = 0)) && ((a3 <= -1) && ((a0 <= 1) && ((a0 >= 1) && ((a4 = 1) && ((a5 =
1) && (-1 <= (a6 + (-1 * a7))))))))))) || (((a1 = a2) && ((a0 = 0) && ((not (a3 = 0)) && ((not (a5 = 0)) && (not (a4 = 0)))))) || (((not (a4 = 0)) && ((a0 = 0) && ((not (a5 = 0)) &&
((a1 = a2) && ((not (a5 = 1)) && ((-4 <= ((-1 * a7) + a6)) && ((4 <= (a7 + (-1 * a6))) && (not (a4 = 1)))))))))) || (((not (a3 = 3)) && ((a3 = 1) && ((a4 = 0) && ((((not (a5 <= 3))
&& ((a4 = 0) && ((a2 = 1) && (a3 = 1)))) || (((not (a5 <= 3)) && ((a4 = 0) && ((not (a2 = 1)) && (((a2 >= 1) || (not (a1 >= 0))) && ((a1 >= 0) && ((a3 = 1) && ((not (a1 <= 0))
&& (not (a2 = 0)))))))))) || ((not (a5 <= 3)) && ((a4 = 0) && ((not (a1 >= 0)) && (a3 = 1)))))) && ((not (a0 >= 4)) && (a5 <= 4)))))) || (((not (a0 = a5)) && (((a2 = 2) || (not (a1 =
a4))) && (((not (((-1 * a6) + a1) <= 1)) || ((a2 = a7) || (not ((a6 + (-1 * a1)) <= -1)))) && ((not (a0 = 3)) && ((not (a5 = 0)) && ((not (a4 = 1)) && ((a5 = 1) && ((not (a3 = 0)) &&
((not (a4 = 2)) && (a4 = 0)))))))))) || (((a5 = 1) && ((not (a0 = 2)) && ((a3 = 2) && ((-1 <= ((-1 * a2) + a1)) && ((1 <= (a2 + (-1 * a1))) && (a4 = 0)))))) || (((a3 = 3) && ((not (a0
= 3)) && (a5 = 1))) || (((a4 = 2) && ((a5 = 2) && ((not (a0 = 0)) && ((a3 <= 1) && ((4 <= (a7 + (-1 * a6))) && ((not (a0 >= 3)) && ((2 <= a0) && ((not (a0 <= 1)) && ((2 <= (a2
+ (-1 * a1))) && ((a0 <= 2) && (((a7 + (-1 * a6)) <= 5) && (((a7 + (-1 * a6)) >= 2) && ((2 <= ((-1 * a1) + a2)) && (-2 <= (a1 + (-1 * a2)))))))))))))))) || (((a0 = 0) && ((not (a4 =
2)) && ((not (a5 = 0)) && ((a1 = a2) && ((not (a4 = 0)) && ((a5 = 1) && (-2 <= (a6 + (-1 * a7))))))))) || (((not (a4 = 0)) && ((not (a5 = 0)) && ((a4 = a5) && ((a3 <= 1) && ((4
<= ((-1 * a6) + a7)) && ((a0 = 1) && ((a4 >= 1) && (((a7 + (-1 * a6)) >= 2) && ((a3 = 0) && ((a5 = 2) && (a4 = 2))))))))))) || (((not (a4 <= 1)) && ((not (a0 = 0)) && ((a3 = 0)
&& ((a0 <= 1) && ((1 <= a0) && ((a0 >= 1) && (3 <= a4))))))) || (((not (a0 = 1)) && ((a4 = 0) && ((not (a5 = 0)) && ((a0 = 0) && ((not (a5 = 2)) && (a1 = a2)))))) || (((not (a4 =
0)) && ((not (a4 = 2)) && ((a0 = 1) && ((not (a3 = 1)) && ((not (a5 = 1)) && ((not (a5 = 0)) && ((a2 = a7) || ((not ((a6 + (-1 * a1)) <= -2)) || (not (((-1 * a6) + a1) <= 2)))))))))) ||
(((a0 = 0) && ((a1 = a2) && ((not (a5 = 2)) && ((not (a4 = 0)) && (not (a4 = 1)))))) || (((a5 = 0) && ((a0 = 1) && ((not (a4 = 1)) && (((a1 <= -1) || (not (a2 = 0))) && ((not (a3 =
1)) && ((not (a4 = 0)) && (0 <= a4))))))) || (((not (a5 = 0)) && ((a4 = 0) && ((a5 = 2) && (((not (a0 <= 2)) || (not (a0 >= 2))) && ((not (a3 = 3)) && ((not (a3 = 1)) && (not (a0 =
0)))))))) || (((not (a5 = 2)) && ((not (a4 = 1)) && ((not (a4 = 2)) && ((a0 = 1) && ((not (a5 = 1)) && ((a4 = 0) && ((not (a3 = 1)) && (a3 = 0)))))))) || (((a5 = 1) && ((not (a0 = 0))
&& ((a4 = 0) && ((a3 = 0) && ((a2 = a7) || (not (a1 = a6))))))) || ((not (a4 = 0)) && ((not (a0 = 1)) && ((a0 = 0) && ((a1 = a2) && ((not (a4 = 2)) && (not (a5 =
1))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

Figure 5. A sample unsimplified interpolant formula from one of the *Pequod* benchmarks.

```
((((((((((((1 + a1) <= a2) && (((a0 != 0) && (((((a0 < 2) && (2 = a3)) && (1 = a4)) && (0 = a5)) || ((((3 = a3) && (2 = a5)) && (1 = a0)) && (0 = a4)))) || (((a3 <= 0) && (a4 !=
0)) && (((((((3 <= a4) && ((5 + a6) <= a7)) && (1 < a5)) && (1 < a4)) && (0 != a5)) && (a0 != 0)) && (1 = a0)) || (((((((1 <= a0) && (0 <= a0)) && (-1 < a3)) && (a0 < 2))
&& (a4 != 1)) && (0 = a5)) && ((a7 + a1) = ((1 + a6) + a2))))))) || ((a4 != 0) && ((((0 != a5) && ((((1 <= a0) && (((((a3 <= -1) && ((4 + a6) <= a7)) && (1 < a4)) && (-2 < a3))
|| (((((((a3 <= 1) && ((6 + a6) <= a7)) && ((1 + a1) <= a2)) && (1 < a5)) && (a0 < (1 + a4))) && (a0 != 2)) && (3 = a5)))) || (((((3 + a6) <= a7) && (2 = a4)) && (-1 = a3)) &&
(((1 <= a0) && (2 = a5)) || (((1 != a5) && (1 != a3)) && (0 = a0))))) || ((((((2 <= a5) && (a3 <= -2)) && ((2 + a6) <= a7)) && ((1 + a1) <= a2)) && (a0 < (1 + a4))) && (a0 !=
0)))) || ((((1 = a5) && (0 = a0)) && (a2 = a1)) && (((a3 <= -1) && (1 < a4)) || ((1 = a4) && (a7 = (3 + a6)))))) || ((((((-1 < a3) && (a4 != 1)) && (a0 != 0)) && (1 = a3)) && (1 =
a0)) && (0 = a5))))) || ((2 = a4) && ((((a4 != 0) && (0 = a5)) && (((0 = a0) && ((a3 <= -1) || (((2 <= a3) && ((3 + a6) <= a7)) && (a4 != 1)))) || ((((a3 <= -1) && ((1 + a6) <=
a7)) && ((1 + a1) <= a2)) && (1 = a0)))) || ((((((1 <= a0) && (a0 < 2)) && (1 != a5)) && (0 != a5)) && (a5 != 2)) && (3 = a5))))) || ((1 = a5) && ((((a0 <= 1) && (3 = a3)) || (((1
<= a0) && (1 = a3)) && (0 = a4))) || (((0 = a0) && (a2 = a1)) && ((((a7 <= (1 + a6)) && (a3 <= -1)) && (1 = a4)) || ((((3 <= a3) && (0 != a5)) && (a5 != 2)) && (0 = a4))))))) ||
((1 = a4) && ((((((a0 != 0) && ((((3 <= a5) && ((1 + a1) <= a2)) && (a0 < 2)) || ((((a3 != 2) && (1 = a0)) && (0 = a5)) && (0 = a3)))) || ((0 = a5) && ((((1 != a5) && ((((1 <=
a0) && (a3 <= -1)) || (((3 <= a3) && ((3 + a6) <= a7)) && (0 = a0))) || (((2 <= a0) && (a3 != 2)) && (0 = a3)))) || (((1 <= a0) && (a3 <= 1)) && (((1 + a6) != a1) || (a7 = a2))))))
|| ((0 = a0) && ((((0 < a3) && ((((3 <= a5) && (a3 <= 1)) && ((6 + a6) <= a7)) || ((((2 + a6) <= a7) && (0 = a5)) && (a2 = a1)))) || ((3 = a5) && (3 = a3))) || ((a2 = a1) && ((((3
<= a5) && (3 = a3)) || ((2 <= a3) && (1 = a5))) || ((((a3 <= 1) && ((4 + a6) <= a7)) && (1 != a5)) && (3 = a5))))))) || ((-1 = a3) && (((1 != a5) && (0 = a0)) || ((0 != a5) && ((((1
<= a0) && (0 <= a0)) && (1 = a5)) || ((((((2 + a6) <= a7) && ((1 + a1) <= a2)) && (a0 != 0)) && (2 = a5)) && (1 = a0))))))) || (((((((1 <= a0) && (a3 <= (1 + a0))) && ((1 + a1)
<= a2)) && (a0 < 2)) && (a0 < a3)) && (0 != a5)) && (2 = a5))))) || ((1 = a0) && (((((((0 <= a0) && (a3 <= -1)) && ((1 + a1) <= a2)) && (3 = a4)) && (0 = a5)) && ((a7 + a1)
= ((1 + a6) + a2))) || ((((((1 != a5) && (0 != a5)) && (1 = a4)) && (1 = a3)) && (a7 = (4 + a6))) && (a2 = (1 + a1)))))) || ((0 = a5) && ((((1 != a5) && ((((0 = a4) && (0 = a3))
&& (a2 = (1 + a1))) || (((((1 <= a0) && (a3 <= -2)) && (a4 != 0)) && (a3 != 0)) && (a7 = a6)))) || ((0 = a0) && ((((4 <= a4) && (a3 <= -1)) || (((2 + a6) <= a7) && (((2 = a3) &&
(1 = a4)) || ((((1 <= a4) && (a4 != 1)) && (1 = a3)) && (a2 = a1))))) || ((a7 = a6) && ((((a3 <= -1) && (a4 != 1)) && ((1 != a1) || (0 != a2))) || (((a3 <= -2) && (a4 != 2)) && (a4 !
= 0))))))) || ((((0 <= a4) && (0 <= a0)) && (a3 <= -2)) && (a4 != 0))))) || ((0 = a4) && (((((((2 = a5) && (((((1 != a3) && (1 != a0)) && (0 != a5)) && (a3 != 3)) || (((((a7 <= (2 +
a6)) && (a0 != 0)) && (1 = a0)) && (0 = a3)) && (a2 = (1 + a1))))) || ((0 = a5) && ((((a0 != 0) && ((((a7 <= (1 + a6)) && (a3 <= 1)) && ((a1 < 1) || (0 != a2))) || (((((0 <= a0)
&& (a7 <= (2 + a6))) && ((1 + a1) <= a2)) && (1 != a5)) && (((1 + a6) != a1) || (a7 = a2))))) || (((1 < a3) && (0 = a0)) && (a2 = a1))) || ((((((a0 <= 1) && ((3 + a6) <= a7)) &&
((1 + a1) <= a2)) && (1 != a3)) && (3 = a3)) && ((a1 <= -1) || (0 != a2)))))) || ((0 = a0) && ((((3 <= a5) && ((1 < a3) || (((((3 + a6) <= a7) && (a4 != 1)) && (0 = a3)) && (a7 =
(a6 + a5))))) || ((1 != a5) && ((3 = a3) || ((2 = a3) && ((1 != a1) || (0 != a2)))))) || ((a2 = a1) && (((3 = a5) && (1 = a3)) || ((2 = a3) && (1 = a5))))))) || ((a2 = a1) && ((((a3 <= 1)
&& (0 = a5)) && (a7 = (1 + a6))) || ((((a0 != 3) && (4 = a5)) && (1 = a3)) && ((1 != a1) || (0 != a2)))))) || ((a2 = (1 + a1)) && ((((a0 <= 1) && (3 = a5)) && (2 = a3)) || ((((1 <=
a0) && (a0 != 2)) && (1 = a5)) && (a7 = (3 + a6)))))) || (((((1 + a1) <= a2) && (3 = a5)) && (1 = a0)) && ((3 = a3) || (0 = a3)))))) || ((0 = a3) && (((((1 = a0) && (((3 <= a4) &&
(0 = a5)) || ((3 = a4) && (1 = a5)))) || ((0 = a4) && (((1 = a5) && (((a2 = a1) && (((1 + a6) != a1) || (a7 = a1))) || (((0 != a5) && (a5 != 2)) && (a0 != 0)))) || ((((((4 + a1) <= (a2 +
a0)) && (a5 != 2)) && (a0 != 0)) && (0 = a5)) && (a7 = a6))))) || ((0 = a0) && (((2 <= a4) && (a4 != 2)) || (((1 = a4) && (a2 = a1)) && ((((2 + a6) <= a7) && (1 = a5)) || ((2 =
a5) && (((3 + a6) != a1) || (a7 = a1)))))))) || (((a0 != 0) && (1 = a4)) && (((0 <= a0) && (1 = a5)) || (((1 != a5) && (2 = a5)) && (a2 = (1 + a1))))))) || ((0 = a0) && (((((((2 <=
a4) && (((3 <= a5) && (((a3 <= 1) && ((6 + a6) <= a7)) || ((a3 <= -1) && ((4 + a6) <= a7)))) || ((((a3 <= 1) && ((5 + a6) <= a7)) && (3 = a5)) && (a2 = a1)))) || ((1 = a4) &&
(((3 <= a5) && (2 = a3)) || (((2 = a5) && (a2 = a1)) && ((2 = a3) || ((0 != a5) && (3 = a3))))))) || ((0 = a3) && (((((3 + a6) <= a7) && (3 = a5)) && (1 = a4)) || (((a4 != 1) && (0
= a5)) && (a7 = a6))))) || ((a2 = a1) && ((((((a4 != 0) && ((((((1 <= a4) && (a3 <= -1)) && ((2 + a6) <= a7)) && (3 = a5)) || ((((0 <= a4) && (a3 <= 0)) && (a4 != 1)) && (0 =
a5))) || ((((a3 <= 1) && (0 < a3)) && (a4 != 2)) && (2 = a5)))) || ((2 = a4) && (((2 = a5) || ((2 <= a5) && ((4 + a6) <= a7))) || (((1 <= a3) && (1 < a4)) && (1 = a5))))) || ((1 = a5)
&& (((2 <= a4) && (a4 != 2)) || (((1 < a4) && (a3 != -1)) && (a7 = (3 + a6)))))) || ((1 = a4) && ((((4 <= a5) && ((4 + a6) <= a7)) && (0 = a3)) || (((a3 <= 0) && (0 = a5)) &&
(a7 = (1 + a6)))))) || ((0 = a4) && ((((a5 <= 2) && (1 < a5)) && (((3 + a6) != a1) || (a7 = a1))) || ((((a3 != 0) && (a3 != -1)) && (1 = a5)) && (((2 + a6) != a1) || ((a7 = a1) && (a2
= a1))))))))) || (((3 <= a4) && (a3 <= -1)) && (2 = a5))) || (((2 <= a5) && (2 <= a3)) && (2 = a4))))) || ((a2 = (1 + a1)) && (((a0 != 0) && (((((((1 <= a0) && (a3 <= 0)) && (a0 <
2)) && (2 = a5)) && (2 = a4)) && (a7 = (4 + a6))) || (((((((1 <= a4) && (a4 <= 2)) && ((3 + a6) <= a7)) && (1 < a4)) && (a4 != 0)) && (1 = a5)) && (1 = a0)))) || ((1 = a0) &&
((((a4 != 0) && (a3 != 0)) && (1 = a5)) || ((((0 != a5) && (a0 != 0)) && (1 = a3)) && (0 = a4))))))) || (((0 != a5) && (2 = a5)) && ((((((a3 <= 1) && ((1 + a1) <= a2)) && (2 =
a4)) && (1 = a0)) && (a7 = (5 + a6))) || ((((((1 <= a0) && (a0 != 0)) && (2 = a3)) && (0 = a4)) && (a7 = (4 + a6))) && (a2 = (1 + a1)))))
```

Figure 6. A sample simplified interpolant formula.

REFERENCES

[1] Strichman, O.: 'Regression Verification: Proving the Equivalence of Similar Programs'. Proc. Proceedings of the 21st International Conference on Computer Aided Verification, Grenoble, France2009 pp. Pages

[2] Kundu, S., Tatlock, Z., and Lerner, S.: 'Proving optimizations correct using parameterized program equivalence', SIGPLAN Not., 2009, 44, (6), pp. 327-337

[3] McMillan, K.L.: 'Lazy Abstraction with Interpolants', in Ball, T., and Jones, R.B. (Eds.): 'Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings' (Springer Berlin Heidelberg, 2006), pp. 123-136

[4] McMillan, K.L., and Rybalchenko, A.: 'Solving Constrained Horn Clauses using Interpolation', in Editor (Ed.)^(Eds.): 'Book Solving Constrained Horn Clauses using Interpolation' (2013, edn.), pp.

[5] Moura, L.D., Bj, N., #248, and rner: 'Z3: an efficient SMT solver'. Proc. Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, Budapest, Hungary, 2008, pp. Pages

[6] Barthe, G., Crespo, J.M., and Kunz, C.: 'Relational Verification Using Product Programs', in Editor (Ed.)^(Eds.): 'Book Relational Verification Using Product Programs' (Springer, 2011, edn.), pp. 200-214

[7] Zhou, Q., Heath, D., and Harris, W.: 'Completely Automated Equivalence Proofs', 2017

[8] T. Lampert, "Minimizing disjunctive normal forms of pure first-order logic," Logic Journal of the IGPL, vol. 25, no. 3, pp. 325–347, 2017.

[9] Zhou, Q., and Harris, W.: 'Solving Constrained Horn Clauses Using Dependence-Disjoint Expansions', CoRR, 2017, abs/1705.03167