Revisiting Positive Equality

Shuvendu K. Lahiri Randal E. Bryant Amit Goel
Muralidhar Talupur

November 2003

CMU-CS-03-196

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper provides a stronger result for exploiting positive equality in the logic of Equality with Uninterpreted Functions (EUF). Positive equality analysis is used to reduce the number of interpretations required to check the validity of a formula. We remove the primary restriction of the previous approach proposed by Bryant, German and Velev [5], where positive equality could be exploited only when all the function application for a function symbol appear in *positive* context. We show that the set of interpretations considered by our analysis of positive equality is a subset of the set of interpretations considered by the previous approach. The paper investigates the obstacles in exploiting the stronger notion of positive equality (called *robust positive equality*) in a decision procedure and provides a solution for it. We present empirical results on some hardware and software verification benchmarks.

This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.

Keywords: Decision Procedures, Logic of Equality with Uninterpreted functions, Verification

1 Introduction

Decision procedures for quantifier-free First-Order Logic (FOL) with equality have become an integral part of many formal verification tools. The importance of decision procedures lies in automatically validating (or invalidating) formulas in the logic. The ability to automatically decide formulas has been the corner-stone of several scalable verification approaches. For hardware, Burch and Dill [9] have used symbolic simulation with a decision procedure for the quantifier-free fragment of FOL to automatically verify complex microprocessor control. Bryant et al. [6, 23] have extended their method to successfully verify superscalar and VLIW processors. Recently, Lahiri, Seshia and Bryant [15] have demonstrated the use of efficient decision procedures to improve the automation for out-of-order processor verification. For software, decision procedures have been used for translation validation of compilers [19]. Decision procedures are used extensively for predicate abstraction in several software verification efforts [2, 12, 14]. They have also been used for the analysis of other concurrent infinite-state systems.

Most decision procedures for quantifier-free logic fall roughly into two categories: decision procedures based on (i) a Combination of Theories [22, 17, 3, 18] or (ii) a validity preserving translation to a Boolean formula [5, 19, 21, 8]. The former methods combine the decision procedures for individual theories using Nelson-Oppen [17] style of combination. The latter methods translate the first-order formula to a Boolean formula such that the Boolean formula is valid if and only if the first-order formula is valid. There has also been work in solving first-order formulas by using abstraction-refinement based on Boolean Satisfiability (SAT) solvers [4, 10].

Among the decision procedures based on a validity preserving translation to a Boolean formula, Bryant et al. [5, 6] proposed a technique to exploit the structure of equations in a formula to efficiently translate it into a Boolean formula. Their method identifies a subset of function symbols in the formula as "p-function" symbols, the function symbols which only occur in monotonically positive contexts. The method then restricts the set of interpretations for the function applications of p-function symbols for checking the validity of the formula. They have successfully used this decision procedure to automatically verify complex microprocessors. The method was initially proposed for the Logic of Equality with Uninterpreted Functions (EUF) and was later extended for the logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU) [8, 13]. Pnueli et al. [19] use Ackermann's function elimination method [1] to remove function applications from a formula and allocate ranges for each of the variables in the resulting formula, such that the ranges are sufficient for checking validity. The technique also exploits the polarity of equations in the formula to restrict the range allocation. Rodeh et al. [21] have used the function elimination method of Bryant et al. [5] to further restrict the domain size of the variables using the algorithm in [19]. The last two decision procedures have been successfully used for validating compiler code automatically. In all the above decision procedures [5, 19, 21], the key idea has been to restrict the set of interpretations, by exploiting the polarity of the terms in the formula.

One of the main limitations of the *positive equality* analysis of Bryant et al. is that it is not *robust*. For a function symbol f to be a "p-function" symbol, *all* the function applications of f have to appear in monotonically positive equations. This makes it difficult to exploit positive equality, even when a small number of applications of a function appears in a *negative* context. This places stronger restrictions on the formulas to be decided efficiently and the method has not proven effective for benchmarks which display these characteristics [20].

In this paper, we present a generalization of positive equality analysis of Bryant, German and

Velev [5] which allows the decision procedure to exploit positive equality in situations where the previous approach could not exploit it. This stronger version of positive equality analysis, called *robust* positive equality, restricts the interpretations to consider in deciding formulas in EUF to a *subset* of interpretations considered by the previous approach. We show the complexity of exploiting *robust* positive equality in a decision procedure which uses the function elimination method proposed by Bryant et al. [6]. We describe a decision procedure to exploit this stronger form of positive equality. We present verification benchmarks where this approach reduces the number of interpretations to consider by orders of magnitude compared to the previous approach.

The rest of the paper is organized as follows: In Section 2, we present Bryant et al.'s positive equality analysis. We illustrate the strengths and limitations of their approach. In Section 3, we present a generalization of the positive equality analysis called *robust* positive equality analysis. We present the *robust maximal diversity* theorem that allows us to restrict the interpretations to consider to be a subset of the interpretations considered by the previous approach. Section 4 discusses a decision procedure based on robust positive equality. We discuss the main complications in exploiting robust positive equality in a decision procedure and provide a heuristic which lets us exploit the robust positive equality. In Section 5, we compare the effectiveness of the new approach compared to the previous work on a set of verification benchmarks.

2 Background: Positive Equality and its Limitation

In earlier work, Bryant et al. [5, 6] exploited *positive equality* in the logic of EUF to give a very efficient decision procedure for this fragment. The logic of EUF is built from *terms* and *formulas*. Terms are formed by function applications (e.g. $f(x)$) or by *if-then-else* (*ITE*) constructs. The expression $ITE(G, T_1, T_2)$ selects T_1 when G is **true**, and T_2 otherwise. Formulas are built from predicate applications, equations between terms or using the other Boolean connectives ($\wedge$, $\vee$, $\neg$). Every function and predicate symbol has an associated arity to denote the number of arguments for the function. Function symbols of arity zero are called *symbolic constants*. Similarly, predicate symbols of arity zero are called *propositional symbolic constants*.

In positive equality analysis, the decision procedure partitions the function symbols in an EUF formula as p-function symbols and g-function symbols. A function symbol f is called a p-function symbol in an EUF formula F[1], if none of the function applications of f appear in (i) a negative equation (e.g. $f(x_1, \ldots, x_k) \neq T_1$) or (ii) in the controlling formula of an *if-then-else* (*ITE*) term (the controlling formula of an *ITE* is implicitly negated when choosing the *else* branch). All function symbols which are not p-function symbols are g-function symbols.

The semantics of an expression in EUF is defined relative to a non-empty domain $\mathcal{D}$ of values and an interpretation I, which assigns values to the function and predicate symbols in the formula. An interpretation I assigns a function from $\mathcal{D}^k$ to $\mathcal{D}$ for each function of arity k and a function from $\mathcal{D}^k$ to $\{$**true**,**false**$\}$ for each predicate symbol of arity k. Given an interpretation I, the meaning of an expression E is defined as $I[E]$ inductively on the syntactic structure of E. A formula F is *valid* (also called *universally valid*), if for every interpretation I, $I[E] = \textbf{true}$.

An interpretation I is called a *maximally-diverse* interpretation[2], if for any *p-function* symbol f,

[1]For simplicity, assume F is in negation normal form where all the negations are pushed down towards the leaves of the formula and $\neg\neg G$ is collapsed to G.

[2]The definition of maximally-diverse interpretation is slightly different from the original work [5] for simplicity of presentation.

$I[f(U_1, \ldots, U_k)] = I[g(S_1, \ldots, S_m)]$ if and only if the following conditions hold: (i) f and g are the same function symbol and (ii) forall $i \in [1, \ldots, k]$, $I[U_i] = I[S_i]$. The main theorem is called the *maximal diversity theorem*, which is given below.

Theorem 1 Maximal Diversity Theorem. *An EUF formula F is valid iff F is true in all maximally-diverse interpretations.*

Restricting the set of interpretations to only maximally-diverse interpretations for checking validity is very efficient for EUF formulas with large number of p-function symbols. For instance, consider the formula:

$$\neg(x = y) \vee f(g(x)) = f(g(y))$$

The set of terms in the formula is $\{x, y, g(x), g(y), f(g(x)), f(g(y))\}$. Since there are 6 terms in the formula, it is sufficient to restrict the domain of each of the terms to contain at most 6 values, for checking the validity [1]. Hence, one can decide the formula by considering 6^6 interpretations. However, positive equality analysis allows us to restrict the number of combinations to search, to only 2^2 values, since only two functions x and y (of arity 0) appear in a negative equation.

However, the main bottleneck of the approach is that it is not *robust*. Positive equality can not be exploited for a function symbol f even if only one application of f appears in a negative context. For example, consider the following EUF formula:

$$F \doteq \neg(f(x) = x) \vee (f(f(f(f(x)))) = f(f(f(x)))) \quad (1)$$

After exploiting positive equality, the set of p-function symbols would be {} and the set of g-function symbols would be {x,f}. This is because both x and f appear in a negative equation, namely $\neg(f(x) = x)$ in the formula. Thus the number of interpretations to search would be $5^5 = 3125$.

However, one can see that only one application of f, namely $f(x)$, appears in a negative equation while the other applications, $f(f(x))$, $f(f(f(x)))$ and $f(f(f(f(x))))$, appear in positive equations only. In this paper, we present a generalization of the positive equality analysis which allows us to exploit the positive structure of such applications. Based on the new analysis, it is sufficient to consider only 4 interpretations to decide the validity of the formula F, instead of the 5^5 interpretations. Even for this small formula, this reduces the number of interpretations to consider $3125/4 = 781$ fold !

3 Logic of Robust Positive Equality with Uninterpreted Functions (RPEUF)

3.1 Syntax

Figure 1 gives the syntax of RPEUF[3]. The logic is essentially same as EUF or PEUF [5], but partitions the formulas (respectively, terms) into "p-formulas" and "g-formulas" (respectively, "p-terms" and "g-terms"). Intuitively, a p-formula appears in only monotonically positive expressions,

[3]We try to follow the terminology of the original paper by Bryant et al. for the rest of the paper, whenever applicable

i.e. does not appear under the scope of negations ($\neg$), or in the controlling formulas of *ITE* expressions. All other formulas are g-formulas. The top-level formula can always be classified as a p-formula. The p-terms are those terms which never appear in a g-formula. More details can be found in [6]. The only difference between PEUF and RPEUF is that function symbols are not partitioned as p-function symbols and g-function symbols. Instead, each application of functions can either be a p-function application (*p-func-appl*) or a g-function application (*g-func-appl*). Let $\mathcal{T}_p(F)$ be the set of p-term function application terms in a formula F. Similarly, let $\mathcal{T}_g(F)$ be the set of g-term function application terms in a formula F.

$$
\begin{array}{rcl}
\textit{g-term} & ::= & \textit{ITE}(\textit{g-formula},\ \textit{g-term},\ \textit{g-term}) \\
 & | & \textit{g-func-appl}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{p-term} & ::= & \textit{g-term} \mid \textit{ITE}(\textit{g-formula},\ \textit{p-term},\ \textit{p-term}) \\
 & | & \textit{p-func-appl}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{g-formula} & ::= & \textbf{true} \mid \textbf{false} \mid \neg \textit{g-formula} \mid (\textit{g-term} = \textit{g-term}) \\
 & | & (\textit{g-formula} \vee \textit{g-formula}) \mid (\textit{g-formula} \wedge \textit{g-formula}) \\
 & | & \textit{predicate-symbol}(\textit{p-term}, \ldots, \textit{p-term}) \\
\textit{p-formula} & ::= & \textit{g-formula} \mid (\textit{p-term} = \textit{p-term}) \\
 & | & (\textit{p-formula} \vee \textit{p-formula}) \mid (\textit{p-formula} \wedge \textit{p-formula})
\end{array}
$$

Figure 1: Syntax for RPEUF

For any RPEUF formula F, we define $\Sigma(F)$ to be the set of function symbols in F. For a function application term T, *top-symbol*(T) returns the top-level function symbol for the term T.

3.2 Diverse Interpretations

The semantics of an expression in RPEUF is defined in a similar manner as defined in Section 2. The domain $\mathcal{D}$ is kept implicit for most of our purposes and we assume it to be the underlying domain. An interpretation defines a partitioning of the terms in the formula, where two terms belong to the same equivalence class if and only if they are assigned the same value. Interpretation I *refines* (*properly refines*) interpretation I', if I refines (properly refines) the equivalence classes induced by I'.

Given an interpretation I, function application terms $T_1 \doteq f(U_1, \ldots, .U_k)$ and $T_2 \doteq f(S_1, \ldots, S_k)$ are said to *argumentMatch* under I, if for all $j \in [1, \ldots, k]$, $I[U_j] = I[S_j]$. It is not defined when T_1 and T_2 have different top-level function symbols.

Robust Maximally Diverse Interpretation. An interpretation I is said to be *robust maximally diverse* if I satisfies the following property:

- For every term $T_1 \doteq f(U_1, \ldots, U_k) \in \mathcal{T}_p(F)$, which does not *argumentMatch* under I with any term $f(S_1 \ldots S_k) \in \mathcal{T}_g(F)$, and for any other function application term T_2, $I[T_1] = I[T_2]$ iff (i) $T_2 \doteq f(V_1, \ldots, V_k)$, and (ii) $I[U_m] = I[V_m]$, for all $m \in [1 \ldots k]$.

3.2.1 Example

Consider the formula in Equation 1. The interpretation Consider the formula in Equation 1. Let us assume (shown a little later in Section 4.1), the set $\mathcal{T}_p(F) \doteq \{f(f(x)), f(f(f(x))), f(f(f(f(x))))\}$, the set of positive applications. The set $\mathcal{T}_g(F)$ becomes $\{x, f(x)\}$. The interpretation $I \doteq \{x \mapsto 1, f(1) \mapsto 2, f(2) \mapsto 3, f(3) \mapsto 4\}$ is an example of a robust maximally diverse interpretation. In this interpretation, $I[f(x)] = 2, I[f(f(x))] = 3$ and $I[f(f(f(x)))] = 4$. Similarly, the interpretation $I \doteq \{x \mapsto 1, f(1) \mapsto 2, f(2) \mapsto 2\}$ is a robust maximally diverse interpretations. However, the interpretation $I \doteq \{x \mapsto 1, f(1) \mapsto 2, f(2) \mapsto 1\}$ is not a robust maximally diverse interpretation since $I[x] = I[f(f(x))] = 1$. But $f(f(x))$ is a *p-term*, whose argument $I[f(x)] = 2$ does not match the argument of the *g-term* $f(x)$, since $I[x] = 1$.

Theorem 2 Robust Maximal Diversity Theorem. *A p-formula F is universally valid iff F is true in all robust maximally diverse interpretations.*

The theorem allows us to restrict ourselves to only those interpretations which are robust maximally diverse. We will show later that in many cases, this prunes away a very large portion of the search space. The proof is very similar to the one presented for the maximal diversity theorem [6], and thus we present it in the appendix of the paper.

The following lemma establishes the correspondence between the maximally diverse interpretations and the robust maximally diverse interpretations.

Proposition 1 *If an interpretation I is a robust maximally diverse interpretation, then I is a maximally diverse interpretation.*

This follows from the fact, that for a "p-function" symbol f, a p-term $T_1 \doteq f(U_1, \ldots, U_k)$ never *argumentMatch* with a g-term $T_2 \doteq f(V_1, \ldots, V_k)$, since there are no g-terms for a "p-function" symbol f. Thus the set of robust maximally diverse interpretations is a subset of the set of maximally diverse interpretation set.

4 Decision Procedure for Robust Positive Equality

In this section, we present a decision procedure for exploiting robust positive equality. The essence of the decision procedure is similar to the decision procedure proposed by Bryant, German and Velev. But there are important differences which makes the procedure more complicated.

4.1 Extracting a RPEUF from EUF

Given a EUF formula F, one might try to label the terms and formulas as g-terms, p-terms, p-formulas, g-formulas by the syntax in Figure 1. But the choice of "promoting" g-terms and g-formulas to p-terms and p-formulas makes the grammar ambiguous. Thus the first step is to use a labeling scheme to mark the different expressions in the formula F.

For a given EUF formula F, let $\mathcal{L}_F$ be a labeling function. If $\mathcal{T}(F)$ and $\mathcal{G}(F)$ be the set of terms and formulas in F, then $\mathcal{L}_F$ satisfies the following conditions:

- If $T \in \mathcal{T}(F)$, then $\mathcal{L}_F(T) \in \{g\text{-}term, p\text{-}term\}$
- If $G \in \mathcal{G}(F)$, then $\mathcal{L}_F(G) \in \{g\text{-}formula, p\text{-}formula\}$
- This labeling is permitted by the syntax

A natural labeling function $\mathcal{L}_F^*$ [6] is to label the formulas which never appear under an odd number of negations and does not appear as a control for any ITE node, as *p-formula*. All other formulas are labeled as *g-formula*. Once the formulas are labeled, label a term as *p-term* if it never appears in an equation labeled as *g-formula*. All other terms are marked *g-term*. This simple labeling scheme has the following important property.

Proposition 2 *Let $\mathcal{T}_p^{\mathcal{L}^*}(F)$ be the set of set of p-term function applications in F produced by the labeling scheme $\mathcal{L}_F^*$. For any other labeling scheme $\mathcal{L}_F^1$, and $\mathcal{T}_p^{\mathcal{L}^1}(F)$,*

$$\mathcal{T}_p^{\mathcal{L}^*}(F) \supseteq \mathcal{T}_p^{\mathcal{L}^1}(F)$$

4.2 Topological Ordering of terms

Once we have labeled all the terms in a formula F as either a p-term or a g-term, we will define a *topological order* $\preceq$, for visiting the terms. A topological order preserves the property that if T_1 is a subterm of T_2 in the formula F, then $T_1 \preceq T_2$. There can be many topological orders for the same formula.

Given a topological order $\preceq$, consider the terms that have been "labeled" by $\mathcal{L}(F)$. We will partition the terms into $\mathcal{T}_{\preceq}^{+}(F)$, $\mathcal{T}_{\preceq}^{-}(F)$ and $\mathcal{T}_{\preceq}^{*}(F)$ as follows: For any term $T \in \mathcal{T}(F)$:

- $T \in \mathcal{T}_{\preceq}^{-}(F)$ iff $\mathcal{L}(T) = g\text{-}term$
- $T \in \mathcal{T}_{\preceq}^{*}(F)$ iff $\mathcal{L}(T) = p\text{-}term$ and there exists $T_1 \in \mathcal{T}_{\preceq}^{-}(F)$ such that $T \preceq T_1$ and *top-symbol*(T) = *top-symbol*(T_1).
- $T \in \mathcal{T}_{\preceq}^{+}(F)$ iff $T \notin \mathcal{T}_{\preceq}^{-}(F)$ and $T \notin \mathcal{T}_{\preceq}^{*}(F)$.

Intuitively, the terms in $\mathcal{T}_{\preceq}^{*}(F)$ are those terms which precede a negative application with the same top-level function symbol. We label some terms as members of $\mathcal{T}_{\preceq}^{*}(F)$ because the function elimination scheme (based on Bryant et al.'s method) eliminates function applications in a topological order. Hence we need to process all the subterms before processing a term.

For example, consider the formula in Equation 1. There are 5 terms in the formula: x, $f(x)$, $f(f(x))$, $f(f(f(x)))$, $f(f(f(f(x))))$. The labeling scheme labels the terms $x, f(x)$ as g-term and the terms $f(f(x)), f(f(f(x))), f(f(f(f(x))))$ as p-term. The only topological ordering on this set of terms is $x \preceq f(x) \preceq f(f(x)) \preceq f(f(f(x))) \preceq f(f(f(f(x))))$. Given this topological order, the partitioning results in the following sets

- $\mathcal{T}_{\preceq}^{-}(F) = \{x, f(x)\}$, $\mathcal{T}_{\preceq}^{*}(F) = \{\}$ and $\mathcal{T}_{\preceq}^{+}(F) = \{f(f(x)), f(f(f(x))), f(f(f(f(x))))\}$.

However, consider the following formula:

$$F \doteq \neg(f(g(x)) = g(f(x))) \tag{2}$$

There are 5 terms in the formula: x, $f(x)$, $g(x)$, $f(g(x))$ and $g(f(x))$. The labeling labels $f(g(x)), g(f(x))$ as g-term and $x, f(x), g(x)$ as p-term. Three possible topological orderings on this set of terms are:

1. $x \preceq f(x) \preceq g(x) \preceq f(g(x)) \preceq g(f(x))$, or
2. $x \preceq f(x) \preceq g(f(x)) \preceq g(x) \preceq f(g(x))$, or
3. $x \preceq g(x) \preceq f(g(x)) \preceq f(x) \preceq g(f(x))$

Given these topological order, the partitioning results in the following sets for the three orders, respectively:

1. $\mathcal{T}_{\preceq}^{-}(F) = \{f(g(x)), g(f(x))\}$, $\mathcal{T}_{\preceq}^{*}(F) = \{f(x), g(x)\}$ and $\mathcal{T}_{\preceq}^{+}(F) = \{x\}$.
2. $\mathcal{T}_{\preceq}^{-}(F) = \{f(g(x)), g(f(x))\}$, $\mathcal{T}_{\preceq}^{*}(F) = \{f(x)\}$ and $\mathcal{T}_{\preceq}^{+}(F) = \{x, g(x)\}$.
3. $\mathcal{T}_{\preceq}^{-}(F) = \{f(g(x)), g(f(x))\}$, $\mathcal{T}_{\preceq}^{*}(F) = \{g(x)\}$ and $\mathcal{T}_{\preceq}^{+}(F) = \{x, f(x)\}$.

The example in Equation 2 illustrates several interesting points. First, even though $f(x)$ and $g(x)$ are both labeled as p-term, there is no ordering of terms such *all* the g-term with the top-level symbol f and g precede these two terms. Note that this limits us from exploiting the full power of Theorem 2. Second, the topological ordering can affect the size of the set $\mathcal{T}_{\preceq}^{+}(F)$. The bigger the size of this set, the better the encoding is. Hence, we would like to find the topological ordering which maximizes the size of $\mathcal{T}_{\preceq}^{+}(F)$.

4.3 Maximizing $\mathcal{T}_{\preceq}^{+}(F)$

The problem of obtaining the optimal $\preceq$, which maximizes the size of $\mathcal{T}_{\preceq}^{+}(F)$, turns out to be NP-complete. In this section, we reduce the problem of *maximum independent set* for an undirected graph to our problem.

Let us first pose the problem as a decision problem — is there an ordering $\preceq$ for which the number of terms in $\mathcal{T}_{\preceq}^{+}(F)$ is at least k ? Given an ordering $\preceq$, it is easy to find out the number of terms in $\mathcal{T}_{\preceq}^{+}(F)$ in polynomial time, hence the problem is in NP.

To show that the problem is NP-complete, consider a undirected graph $G \doteq \langle V, E \rangle$, with V as the set of vertices and E as the set of edges. Construct a *labeled and polar* directed acyclic graph (DAG) $D \doteq \langle V', E' \rangle$, where each vertex $v \in V'$ is a tuple (n_v, l_v, p_v), where n_v is the vertex identifier, l_v is a *label* of the vertex, and p_v is the *polarity* of the vertex. The label of a vertex is a function symbol, and the polarity of a vertex can either be (-) *negative* or (+) *non-negative*. It is easy to see that the vertices of D represent the *terms* in a formula, the *label* denotes the top-level function symbol associated with the term and a vertex with a *negative* polarity denotes a g-term.

The DAG D is constructed from G as follows:

- For each vertex v in V, create two vertices v^+ and v^- in V', such that $v^+ \doteq (v^1, v, +)$ and $v^- \doteq (v^2, v, -)$.
- For each edge $(v_1, v_2) \in E$, add the following pair of directed edges in E' — (v_1^+, v_2^-) and (v_2^+, v_1^-).

Finally, given an ordering $\preceq$, $\mathcal{T}^+_\preceq(D)$ contains a subset of those v^+ vertices which do not precede the v^- vertex with the same label v in $\preceq$.

Now, we can show the following proposition:

Proposition 3 *The graph G has a maximum independent set of size k if and only if the DAG D has an ordering $\preceq$ which maximizes the number of vertices in $\mathcal{T}^+_\preceq(D)$ to k.*

The proof of the proposition can be found in the Appendix.

4.4 Heuristic to maximize $\mathcal{T}^+_\preceq(F)$

Since the complexity of finding the optimum $\preceq$ is NP-complete, we outline a greedy strategy to maximize the number of p-terms in $\mathcal{T}^+_\preceq(F)$. We exploit the following proposition:

Proposition 4 *Given an ordering $\preceq_g$ over all the g-term of the formula, one can obtain an ordering $\preceq$ over all the terms in the formula in time linear to the size of the formula, such that the number of terms in $\mathcal{T}^+_\preceq(F)$ is maximum over all possible orderings consistent with the order $\preceq_g$.*

It is not hard to see the validity of the proposition. We can simply walk over $\preceq_g$, and for each g-term T_i, we add all the subterms of T_i, which have not been added already, before T_i to construct $\preceq$. Finally, the terms (namely some p-terms) that are not subterms of any of the g-terms in $\preceq_g$, can be added in any topological order to the end of the order $\preceq$. The number of terms in $\mathcal{T}^+_\preceq(F)$ is the maximum across all possible orderings consistent with $\preceq_g$. This is because we did not have any choice for the p-term which were subterms of some g-term, and all the terms added after the last g-term are members of $\mathcal{T}^+_\preceq(F)$.

Hence, our problem has been reduced to finding the optimum ordering $\preceq_g$ among the g-terms of the formula. The algorithm has the following main steps:

1. A term $T_1 \doteq f(S_1, \ldots, S_k)$ is *potentially positive* iff T_1 is a p-term and T_1 is not a subterm of any other g-term T_2, which has the same top-level function symbol f. For each function symbol f, we compute the number of *potentially positive* function applications of f in the formula.

2. Order the list of function symbols depending on the number of potentially positive terms for each function symbol. The essential idea is that if a function f has n_f potentially positive applications, and if we order all the terms of f independent of the applications of other function symbols, then the number of terms in $\mathcal{T}^+_\preceq(F)$ is at least n_f.

3. For each function symbol f, we order all the g-terms of f by simply traversing the formula in a depth-first manner. This ordering of g-terms is consistent with the topological order imposed by the subterm structure.

4. Finally, we obtain $\preceq_g$, by repeatedly placing all the gterms for each of the functions in the sorted function order. While placing a g-term T_1 for function f, we place all the g-terms for the other function symbols which are subterms of the g-term before T_1 in the order.

4.5 Function and predicate elimination

To exploit the robust positive equality, we eliminate the function and predicate applications from the RPEUF formula using Bryant et al.'s method. For a function symbol f which appears in F, we introduce symbolic constants $vf_i, \ldots, vf_k$, where k is the number of distinct application of f in the formula. Then the i^{th} application of f (in the topological ordering $\preceq$) is replaced by the nested ITE formula, $ITE(a_i = a_1, vf_1, ITE(a_i = a_2, vf_2, \ldots ITE(a_i = a_{i-1}, vf_{i-1}, vf_i)))$. Here a_i is the argument list to the i^{th} function application. We say that the symbolic constant vf_i is introduced while eliminating the i^{th} application of f. The following lemma [6] describes the relationship between the original and the function-free formula. Predicate applications are eliminated similarly.

Lemma 1 *For a RPEUF formula F, the function and predicate elimination process produces a formula $\widehat{F}$ which contains only symbolic constants and propositional symbolic constants, such that F is valid iff the function-free formula $\widehat{F}$ is valid.*

Let $\mathcal{D}$ be the domain of interpretations for F. Let $V_{\preceq}$ be the set of symbolic constants introduced while eliminating the function applications and $V^{+}_{\preceq} \subseteq V_{\preceq}$ be the set of symbolic constants introduced for the terms in $\mathcal{T}^{+}_{\preceq}(F)$. Let $\widehat{F}_p$ be the formula obtained by assigning each variable $v_i \in V^{+}_{\preceq}$ a value z_i, from the domain $\mathcal{D}' \doteq \mathcal{D} \cup \{z_1, \ldots, z_m\}$, where $m = |V^{+}_{\preceq}|$ and all z_i are distinct from values in $\mathcal{D}$. Then we can prove the following theorem:

Theorem 3 *The formula F is valid iff $\widehat{F}_p$ is true for all interpretations over $\mathcal{D}$.*

Proof: We give a very informal proof sketch in this paper. A detailed proof can be obtained very similar to the proof shown in the original paper [6].

Let us consider a robust maximally diverse interpretation I for F. Consider a symbolic constant $vf_i \in V^{+}_{\preceq}$, which results while eliminating the i^{th} application of f (say T_i) in the order $\preceq$. Note that T_i is a p-term application. First, consider the case when T_i *argumentMatch* with some other term T_j, such that $T_j \preceq T_i$. In this case, the value given to vf_i does not matter, as it is never used in evaluating $\widehat{F}_p$. On the other hand, consider the case when T_i does not *argumentMatch* with any term T_j, such that $T_j \preceq T_i$. Since all the g-term for f precede T_i in $\preceq$ (by the definition of $\mathcal{T}^{+}_{\preceq}(F)$), it means that $I[T_i]$ is distinct from the values of other terms, unless restricted by functional consistency (by Theorem 2). But the value of vf_i represents the value of $I[T_i]$, under this interpretation. Hence, we can assign vf_i a distinct value, not present in $\mathcal{D}$.

4.6 Extending Robust Positive Equality to CLU

We can extend our method to the Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU), in the same way proposed in UCLID [13, 8]. The only addition in the logic is the presence of inequalities ($<$) and addition by constant offsets ($+c$). In the presence of $<$, we adopt a conservative approach and say that terms T_1, T_2 appear in negative context (g-term) if they appear in an inequality ($T_1 < T_2$). Similarly, a function application term T_1 is classified as g-term if any term $T_1 + c$ (for any c) appears in negative context. Even these conservative extensions have proved beneficial for verification problems in UCLID.

5 Results

5.1 Simple Example

Let us first illustrate the working of the decision procedure on a simple formula. Consider the following formula:

$$\Psi_1 \doteq (f(f(f(y))) = f(f(y))) \vee (f(f(y)) = f(x)) \vee \neg(x = f(y)) \tag{3}$$

The function symbols in the formula are $\Sigma(\Psi_1) = \{f,x,y\}$. Our heuristic finds the following order $\preceq$, which also happens to be the optimal order:

$$x \preceq y \preceq f(y) \preceq f(x) \preceq f(f(y)) \preceq f(f(f(y)))$$

The sets $\mathcal{T}^-_\preceq(\Psi_1)$, $\mathcal{T}^*_\preceq(\Psi_1)$ and $\mathcal{T}^+_\preceq(\Psi_1)$ are:

$$\mathcal{T}^-_\preceq(\Psi_1) = \{x, f(y)\}, \mathcal{T}^*_\preceq(\Psi_1) = \{\}, \mathcal{T}^+_\preceq(\Psi_1) = \{y, f(x), f(f(y)), f(f(f(y)))\}$$

The resultant formula after eliminating the function symbols using the above procedure would be

$$\widehat{\Psi}_1 \doteq (f^4 = f^3) \vee (f^3 = f^2) \vee \neg(x = f^1) \tag{4}$$

where

$$\begin{aligned}
f^1 &\doteq vf_1 \\
f^2 &\doteq \mathit{ITE}(x = y,\, vf_1,\, vf_2) \\
f^3 &\doteq \mathit{ITE}(f^1 = y,\, vf_1,\, \mathit{ITE}(f^1 = x,\, vf_2,\, vf_3)) \\
f^4 &\doteq \mathit{ITE}(f^3 = y,\, vf_1,\, \mathit{ITE}(f^3 = x,\, vf_2,\, \mathit{ITE}(f^3 = f^1,\, vf_3,\, vf_4)))
\end{aligned}$$

Thus $\widehat{\Psi}_1$ has 6 symbolic constants $\{x, y, vf_1, vf_2, vf_3, vf_4\}$. Based on robust maximal diversity theorem, we can assign *distinct* values to y, vf_2, vf_3, vf_4, since they are introduced while eliminating a function application in $\mathcal{T}^+_\preceq(\Psi_1)$. The rest of the symbolic constants x, vf_1 have to take on 2 values each. Thus, it is sufficient to consider $2^2 = 4$ interpretations to decide the validity of the formula. In fact, it is sufficient to consider 1 value for x and 2 values for vf_1 to decide the validity, since they can either be equal or unequal. Therefore, the number of interpretations to consider is 2 for this case. Alternately, one could use a single Boolean variable to encode the equality $x = vf_1$ [11]. The final propositional formula in this case contains a single Boolean variable[4], and thus requires 2 interpretations.

The above formula was also used as a running example in previous work [19, 21]. Figure 2 illustrates the number of interpretations to be considered by each of the approaches, which try to minimize the number of interpretations to consider.

This brings out an interesting point. Rodeh and Strichman [21] claim that their method subsumes Bryant et al.'s positive equality. But for this example, robust positive equality actually does better than their method. Their algorithm can limit the search to 4 interpretations or 2 interpretations depending on the heuristic for this example, whereas we can reduce it to 2. Hence, robust positive

[4]Usually, more variables are added to express transitivity constraints, but this example does not require any, since there is a single Boolean variable

Approach	Number of Interpretations	Remarks
Bryant, German, Velev	$5^5 = 3125$	Uses Positive Equality. Only p-function symbol is y
Pnueli et al.	16	Range allocation for each term after Ackermann's reduction
Rodeh, Strichman	4 or 2	Range allocation after Bryant et al.'s function elimination Depends on the heuristic
Current Work	2	Robust Positive Equality

Figure 2: Comparison of different methods on the example formula.

equality is not subsumed by their approach. However the two approaches are complementary. Robust positive equality analysis can be used as a preprocessing step before exploiting the range-allocation scheme by Pnueli et al. and Rodeh et al.'s methods. Further, robust positive equality analysis can work with the more general logic of CLU [8], but the methods in [19, 21] are restricted to EUF. It is not clear how to extend the range allocation easily in the presence of $<$ and constant offsets.

5.2 Verification Benchmarks

In this section, we compare our algorithm with the original positive equality algorithm, based on a set of software verification benchmarks generated from Translation Validation of Compilers [19] and device-driver verification in BLAST [12]. We also report our experience with other verification benchmarks including those from microprocessor and cache-coherence protocol verification. All the formulas discussed in this section are valid formulas.

We have integrated the new method in the tool UCLID [8]. All the experiments are run on a 1.7GHz machine with 256MB of memory. For all these experiments, the integer variables in the formula (after function elimination) are encoded using a small-domain encoding method [8]. This method assigns each integer variable a finite but sufficiently large domain which preserves validity of the formula. The final propositional formula is checked using a Boolean Satisfiability (SAT) solver. For our case, we use mChaff [16].

Figure 3 compares the number of terms which can be assigned distinct values (i.e. the number of terms whose range contains a single value) for positive equality (*p-vars*) and the robust positive equality (*robust-p-vars*) algorithms. The column with *potential # of p-vars* denotes an upper bound on the total number of positive terms. This is obtained by simply adding the number of *potentially positive* terms for each function symbol without considering the ordering of terms across different function symbol. This is a very optimistic measure and there may not be any order $\preceq$ for which this can be achieved. The time taken by each approach is also indicated in the table.

For most of the code validation benchmarks, the number of p-terms is larger compared with the earlier work. Similar trend is also observed for the BLAST set of benchmarks. For many of the code validation benchmarks, the increase in the number of positive variables translates into an improvement of the total time taken to check the validity of the formula. This is expected as the new method reduces the number of interpretations to search. However, for a few cases, the new method is almost 10% slower than the original method, even when the number of positive variables are 10% larger. This happens because of the overhead of the robust positive equality analysis.

Our current implementation requires multiple passes over the formula, which can often increase the time required to translate a CLU formula into a Boolean formula. However, the time taken by the SAT solver (mChaff) is almost always smaller with the new method. This is particularly effective, when solving formulas for which the SAT solver time dominates the time to translate to a Boolean formula (e.g. `cv46`).

It is interesting to notice that for most benchmarks (except `cv22`) the total number of robust-p-vars is the same as the maximum possible number of p-vars possible. On one hand this suggests that the heuristic we chose is optimal for all these benchmarks. On the other hand, it shows that there are no occurrence of mutually nested function applications with alternate polarity evident in the example $\neg f(g(x)) = g(f(y))$. For this example, the maximum number of *potentially positive* terms is 4 ($\{x, y, f(y), g(x)\}$), but one can obtain at most 3 in any ordering ($\{x, y, f(y)\}$ or $\{x, y, g(x)\}$). This is because a *potentially positive* application for g appears as a subterm of a g-term for f and vice versa. Our tool gives the correct answer for this example.

We also ran our analysis on a large number of hardware benchmarks ranging from a DLX pipeline, an out-of-order processor with unbounded resources and a directory based cache-coherence protocol [7]. For all these examples, the number of *robust p-vars* generated by our analysis matched the number of *p-vars* generated by positive equality. This is not hard to explain. For the processor benchmarks, designing the models often ensured that the number of *p-vars* was maximized. The cache coherence example does not involve any data-path computations. The uninterpreted functions are used to generate arbitrary process identifiers, which are compared with each process identifier, and hence all the applications end up being negative.

Besides, these benchmarks are generated from symbolic simulation of the hardware designs, where the same transition function is replicated across the different steps, resulting in identical structure for different steps. Hence the polarity of a function application is often the same (either all positive or all negative) across different steps of simulation. For other form of benchmarks, which arise mostly from the symbolic simulation of software [2, 12], the formulas are relatively *shallow*, and have different transition functions for different control locations in a sequential program. For such benchmarks, the new approach is likely to do better.

Benchmark	example	# vars	Positive Equality		Robust Positive Equality		
			#p-vars	Time taken (sec)	# p-vars	potential # p-vars	Time taken (sec)
Code Validation	cv1	17	3	1.58	7	7	1.60
	cv2	4	1	0.34	1	1	0.48
	cv20	21	6	0.40	6	6	0.47
	cv22	101	1	70.84	16	18	45.65
	cv23	101	8	23.06	22	22	15.96
	cv25	101	8	45.93	22	22	21.80
	cv37	13	4	6.40	4	4	6.32
	cv44	38	8	19.75	17	17	7.13
	cv46	70	10	> 1800	28	28	100.50
BLAST	bl7	262	109	241.27	125	125	265.38
	bl8	315	125	454.40	142	142	456.80
	blt3	268	72	11.16	94	94	11.90

Figure 3: **Comparison on Software Verification Benchmarks.** The examples with prefix "cc" denote code validation benchmarks and those with prefix "bl" denotes BLAST benchmarks.

6 Conclusion and future work

In this work, we have presented a generalization of the positive equality analysis. The extension allows us to handle benchmarks for which the positive structure could not be exploited using the previous method. The added overhead for this generalization is negligible as demonstrated on some reasonably large benchmarks. An interesting point to observe in this paper is that most of the proofs and mathematical machineries from the previous work have been successfully reused for our extension.

There are other optimizations that can be exploited beyond the current work. We want to exploit the positive equality for the terms in $\mathcal{T}^*_{\precsim}$, which are subterms of g-terms with the same top-level function symbol. Instead of using distinct values for the symbolic constants which arise from the elimination of these terms, we are investigating the addition of extra *clauses* in the final formula, to prevent the SAT-solver from considering these interpretations. We would also like to use other range allocation methods, after exploiting robust positive equality, to further improve the decision procedure.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* North-Holland, Amsterdam, 1954.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

[3] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, November 1996.

[4] C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 236–249, July 2002.

[5] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV '99)*, LNCS 1633, pages 470–482. Springer-Verlag, July 1999.

[6] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[7] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Deciding CLU Logic formulas via Boolean and Pseudo-Boolean encodings. In *Proc. Intl. Workshop on Constraints in Formal Verification (CFV'02)*, September 2002.

[8] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[9] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80, June 1994.

[10] C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem Proving usign Lazy Proof Explication. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification (CAV 2003)*, LNCS 2725. Springer-Verlag, 2003.

[11] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV '98)*, LNCS 1427, pages 244–255. Springer-Verlag, June 1998.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '02)*, pages 58–70, 2002.

[13] S. K. Lahiri. An efficient decision procedure for the logic of Counters, Constrained Lambda expressions, Uninterpreted Functions and Ordering. Technical report, ECE Department, Carnegie Mellon University, May 2001.

[14] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification (CAV 2003)*, LNCS 2725, pages 141–153. Springer-Verlag, 2003.

[15] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159. Springer-Verlag, Nov 2002.

[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, 2001.

[17] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.

[18] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, June 1992.

[19] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domain instantiations. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 455–469. Springer-Verlag, July 1999.

[20] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The Small Model Property: How Small Can It Be? Information and Computation. *Information and Computation*, 178(1):279–293, 2002.

[21] Y. Rodeh and O. Strichmann. Finite Instantiations in Equivalence Logic with Uninterpreted Functions. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification (CAV '01)*, LNCS 2102, pages 144–154, 2001.

[22] R. E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, 1984.

[23] M. N. Velev and R. E. Bryant. Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions and Branch Predication. In *37th Design Automation Conference (DAC '00)*, June 2000.

A Proof of the Robust Maximal Diversity Theorem (Theorem 2)

We will restrict ourselves to the proof of the non-trivial part of the proof. Clearly, if a *p-formula* is universally valid, it is true is all interpretations, including the robust maximally diverse ones. The following lemmas help us prove our claim:

Lemma 2 *If I is not a robust maximally diverse interpretation for p-formula F, then there is a proper refinement I' s.t. $I'[F] \implies I[F]$.*

We will present the proof of this lemma in detail after the next lemma, to first complete the proof of Theorem 2.

Lemma 3 *For a given RPEUF p-formula F, and a non robust maximally diverse interpretation $I^0 \doteq I$, we can refine it finitely many times to reach a robust-maximally-diverse interpretation I^*, such that $I^*[F] \implies I[F]$.*

This is easy to see, as there are only finite number of function applications in any RPEUF formula. Starting with I^0, one can obtain an interpretation I^1 by lemma 2, such that $I^1[F] \implies I^0[F]$. This process can continue until $I^k \doteq I^*$ is robust-maximally diverse.

Now, we present the proof of Lemma 2.

Proof: Since I is not robust maximally diverse, there is a p-term T_1 of the form $f(U_1, \ldots, U_k)$ which does not *argumentMatch* with any g-term with the same function symbol f, and there is another function application term $T_2 \doteq g(S_1, \ldots, S_k)$ such that $I[T_1] = I[T_2] = z$, and either (a) f is not same as g or (b) $I[U_i] \neq I[S_i]$ for some $i \in [1, \ldots, k]$.

We can define a new interpretation I' in the following way:

- $\mathcal{D}' = \mathcal{D}$ U $\{z'\}$
- Define $h : \mathcal{D}' \to \mathcal{D}$ such that

$$h(x) = \begin{cases} z & \text{if } x = z' \\ x & \text{Otherwise} \end{cases}$$

- Define I' as follows
  1. For the function function symbol f

  $$I'[f(x_1, \ldots, x_k)] = \begin{cases} z' & \text{if forall } i \in [1, \ldots, k],\ h(x_i) = I[U_i] \\ I[f(h(x_1), \ldots, h(x_k))] & \text{Otherwise} \end{cases}$$

  2. for any other function or predicate symbol g

  $$I'[g(x_1, \ldots, x_k)] = I[g(h(x_1), \ldots, h(x_k))]$$

We claim the following properties and prove it by induction on the construction of the formulas.

1. For every g-formula G, $I'[G] = I[G]$

2. For every g-term T, $I'[T] = I[T]$
3. For every p-term T, $h(I'[T]) = I[T]$
4. For every p-formula F, $I'[F] \implies I[F]$
5. $I'[T_1] = z'$ and $I'[T_2] = z$

For the base case :

1. g-formula : $I'[\mathbf{true}] = I[\mathbf{true}]$; $I'[\mathbf{false}] = I[\mathbf{false}]$; $I'[p] = I[p]$
2. g-term : $I'[v] = I[v]$ for any g-term
3. p-term : $h(I'[v]) = I[v]$ for any p-term, actually $I'[v] \neq I[v]$ iff $v = T_1$
4. p-formula : same as g-formula

Consider the induction step:

1. g-formula (G) : There are 3 cases to consider
   (a) $\neg B$, $B \vee C$, where B, C are g-formulas. By Inductive Hypothesis, $I'[B] = I[B]$, $I'[C] = I[C]$. Hence $I'[G] = I[G]$.
   (b) $S_1 = S_2$, where S_1, S_2 are g-terms: By Inductive hypothesis $I'[S_1] = I[S_1]$, $I'[S_2] = I[S_2]$. Hence $I'[G] = I[G]$.
   (c) Predicate-application $p(S_1, \ldots, S_n)$: By Inductive hypothesis

$$\begin{aligned} I'[p(S_1, \ldots S_n)] &= I'[p](I'[S_1], \ldots I'[S_n]) \\ &= I[p](h(I'[S_1]), \ldots) \\ &= I[p](I[S_1], \ldots, I[S_n]) \\ &= I[p(S_1, \ldots, S_n)] \end{aligned}$$

   Hence $I'[G] = I[G]$

2. g-terms (T). We have 2 cases to consider.
   (a) $ITE(G, S_1, S_2)$. By Inductive hypothesis $I'[G] = I[G]$, $I'[S_1] = I[S_1]$, $I'[S_2] = I[S_1]$. Hence $I'[T] = I[T]$.
   (b) g-term : $T \doteq g(S_1 ...... S_k)$. Since we have assumed that T_1 does not *argumentMatch* with any other g-term under I, T_1 could not possibly *argumentMatch* with T under I. Thus by the definition of I':

$$\begin{aligned} I'[g(S_1 ..... S_k)] &= I'[g](I'[S_1], \ldots, I'[S_k]) \\ &= I[g](h(I'[S_1]), \ldots, h(I'[S_k])) \\ &= I[g](I[S_1], \ldots, I[S_k]) \\ &= I[g(S_1, \ldots, S_k)] \end{aligned}$$

3. p-terms (T): We can have 3 cases to consider.

(a) T is a g-term. Clearly $I'[T] = I[T]$, hence $h(I'[T]) = I[T]$.

(b) T is $ITE(G,\, S_1,\, S_2)$. By induction Hypothesis: $I'[G] = I[G]$, $h(I'[S_1]) = I[S_1]$, $h(I'[S_2]) = I[S_2]$. Hence

$$\begin{aligned} h(I'[T]) &= \text{if } I'[G] \text{ then } h(I'[S_1]) \text{ else } h(I'[S_2]) \\ &= \text{if } I[G] \text{ then } I[S_1] \text{ else } I[S_2] \\ &= I[T] \end{aligned}$$

(c) T is p-term $g(S_1, \ldots, S_k)$: We consider 2 subcases.

i. if (i) g is the same symbol as f and (ii) $I[U_i] = I[S_i]$, forall $i \in [1, \ldots, k]$, then $I'[T] = z'$, $h(I'[T]) = z$. Hence $h(I'[T]) = I[T]$.

ii. else $I'[T] = I[T] = h(I'[T])$

4. p-formulas (F): We need to consider 3 cases

(a) F is a g-formula. Clearly $I'[F] \implies I[F]$

(b) F is $G_1 \vee G_2$. Since both G_1, G_2 are p-formulas, by Induction Hypothesis, $I'[G_1] \implies I[G_1]$; $I'[G_2] \implies I[G_2]$. Hence $I'[G] \implies I[G]$.

(c) F is $S_1 = S_2$. Here S_1, S_2 are p-terms. By Inductive hypothesis, we know that if $I'[S]$ is different from $I[S]$, then $I'[S] = z'$, and the definition of h tells us that $I[S] = z$ in that case. Consider the different cases:

i. Both $I'[S_1] = I'[S2] = z'$. Then $I[S_1 = S_2] = \textbf{true} = I'[S_1 = S_2]$.

ii. Neither of them equal z', then $I[F] = I'[F]$

iii. Only one of $I'[S_1]$ or $I'[S_2]$ equals z'. Then $I'[F] = \textbf{false}$ and $I'[F] \implies I[F]$.

5. The claim that $I'[T_1] = z'$ and $I'[T_2] = z$ follows from the definition of I' and the Inductive properties.

$$\begin{aligned} I'[T_1] &= I'[f(U_1, \ldots, U_k)] \\ &= I'[f](I'[U_1], \ldots, I'[U_k]) \\ &= I'[f](h(I'[U_1]), \ldots, h(I'[U_k])) \\ &= I'[f](I[U_1], \ldots, I[U_k]) \\ &= z' \\ I'[T_2] &= z \end{aligned}$$

B Proof of NP-completeness

Proof: First, let us assume that the graph G has a maximum independent set of vertices $\{v_{i_1}, \ldots, v_{i_k}\}$, of size k. We claim that there are is an order $\preceq$ for D, such that number of vertices in $\mathcal{T}^+_{\preceq}(D)$ is at least k. Since there are no edges between any pair of vertices in $\{v_{i_1}, \ldots, v_{i_k}\}$, there are no edges $(v^+_{i_j}, v^-_{i_l})$, $0 \leq j \leq k$ and $0 \leq l \leq k$ in E'. Hence we can create an order $\preceq$, where the vertices $\{v^-_{i_1}, \ldots, v^-_{i_k}\}$ appear before $\{v^+_{i_1}, \ldots, v^+_{i_k}\}$. This is because a vertex v^-_i can only constrain a positive vertex v^+_j to appear before it in the order (only if the edge $(v^+_j, v^-_i) \in E'$), and v^+_j does not constrain any other vertex to appear before it in the order (since no directed edge ends in a v^+

vertex in D). Hence, we can always place the vertices $\{v_{i_1}^+, \ldots, v_{i_k}^+\}$ after $\{v_{i_1}^-, \ldots, v_{i_k}^-\}$ in an order. It is easy to see that $\mathcal{T}_{\preceq}^+(D)$ contains at least the vertices $\{v_{i_1}^+, \ldots, v_{i_k}^+\}$.

On the other hand, let us assume that there is an order $\preceq$, such that $\mathcal{T}_{\preceq}^+(D)$ is the set $\{v_{i_1}^+, \ldots, v_{i_k}^+\}$ of k vertices. Then the maximum independent set for G has at least k vertices. In fact, the set $V_I \doteq \{v_{i_1}, \ldots, v_{i_k}\}$ is an indepedent set. To see why, let us assume that only a proper subset $V_I' \subset V_I$ forms an indepedent set. Let $v_{i_j} \in V_I - V_I'$. There has to be a vertex $v_{i_l} \in V_I'$ such that $(v_{i_j}, v_{i_l}) \in E$. That implies that $(v_{i_j}^+, v_{i_l}^-) \in E'$ and $(v_{i_l}^+, v_{i_j}^-) \in E'$. But that contradicts the fact that both $v_{i_j}^+$ and $v_{i_l}^+$ appear in $\mathcal{T}_{\preceq}^+(D)$.