A Term Equality Problem Equivalent to Graph Isomorphism

David A. Basin
Max-Planck-Institut für Informatik
Saarbrücken, Germany
Email:basin@mpi-sb.mpg.de

Abstract

We demonstrate that deciding if two terms containing otherwise uninterpreted associative, commutative, and associative-commutative function symbols and commutative variable-binding operators are equal is polynomially equivalent to determining if two graphs are isomorphic. The reductions we use provide insight into this result and suggest polynomial time special cases.

Key Words: Automatic Theorem Proving, Computational Complexity, Equality Reasoning, Graph Isomorphism.

1 Introduction

Decision procedures for equality play an important role in automated theorem proving. We examine the complexity of a specific problem in equality reasoning: determining if two terms, which possibly contain associative, commutative, and associative-commutative (AC) functions and commutative variable-binding operators (that is, adjacent operators may be swapped) are equal. We prove that this problem is polynomially equivalent to deciding if two graphs are isomorphic. The exact complexity of graph isomorphism, and hence our problem, is still open. While graph isomorphism is clearly in NP, it has neither been proven NP-complete nor in P [9, 13].

If one removes variable-binding operators, then polynomial AC-equality decision procedures are trivial. For example, if $+$ is an (infix) AC-function then we can normalize a term like $s_1 + s_2 + ... + s_n$ by sorting the s_i and left associating the result. More complex terms (e.g., the s_i themselves contain various associative and commutative functions) may be normalized by a bottom up traversal that left associates and sorts subterms as permitted by associativity and commutativity. Two terms are equal iff their normalized counterparts are identical. This kind of a procedure doesn't work though in the presence of bound variables, as the variable renaming (α-conversion) can, as we shall see, effect normalization.

The problem we study arose in reasoning about hardware devices. A common approach to representing hardware (see [5]) is to represent devices as relations on their external ports; relations are combined via conjunction and wires between internal ports are represented with existential quantification. For example, if $xor(a, b, o)$ represent an exclusive-or gate with inputs a and b and output o then a half adder $sum(a, b, cin, sum)$ may be defined as the relation

$$\exists w.xor(cin, b, w) \wedge xor(w, a, sum).$$

In representing hardware, terms are the formulae of the predicate calculus. But when considering if two terms represent the same circuit it should not matter how internal wires are named or ordered, or how devices are ordered. That is, we consider the equality of these terms modulo associativity and commutativity of $\wedge$, commutativity of $\exists$, and renaming of bound variables.

Consider the following example, where p_1 and p_2 are distinct uninterpreted predicates that are neither associative nor commutative.

$$s = \exists u \exists v (p_1(u, v) \wedge p_1(v, u)) \wedge p_2(u, v) \quad (1)$$
$$t = \exists w \exists x p_1(w, x) \wedge (p_2(x, w) \wedge p_1(x, w)) \quad (2)$$

These terms are equal. If w and x in (2) are renamed to v and u, the renamed term is normalized by sorting quantifiers lexicographically and left associating and sorting the summed terms, then the equality is apparent. The critical point is that if we simply normalized s and t (s, as written, is already in the above described normal form) without renaming w and x, then the normal form of t,

$$\exists w \exists x (p_1(w, x) \wedge p_1(x, w)) \wedge p_2(x, w),$$

would not rename to s. Though simple, this example suggest that perhaps contrary to one's intuition, that our slightly extended equality problem is not likely to be as easy as the binding operator free problem.

Of course, in the above example, simply comparing the two p_2 terms tells us what the proper bound variable renaming should be. In general, the problem is not so simple. The difficulty arises because both subterms and binding operators may be reordered and this obscures what, if any, bound variable renamings would allow rewriting the terms to α-variants. If the binding operators do not commute then equality again is decidable in polynomial time. For example, if λ is a non-commutative variable-binding operator, then we can decide the equality of two terms $t_1 = \lambda x_1.\lambda x_2...., \lambda x_k.s_1$ and $t_2 = \lambda y_1.\lambda y_2...., \lambda y_k.s_2$ where the s_i contain associative and commutative function symbols by renaming each bound y_i by x_i and normalizing t_1 and the renamed t_2.

Our complexity result is similar to that of [3, 14] in that they analyze the complexity of matching problems in the presence of associative and commutative function symbols. Indeed, their result that AC-Matching is NP-complete might suggest that our problem is also. However, their problem has more latitude then ours. For example, the substitutions that they consider need not be variable bijections which is required in our problem. Moreover, it would be very surprising if their NP-complete problems are reducible to our problem which we show to be isomorphism complete. Other work relevant to ours is that of Bachmair and Plaisted on associative commutative term rewriting systems [2] and that of Hullot[12] and Gramlich and Denzinger [10] on associative-commutative matching. However, none of these authors consider complications introduced by variable binding operators.

Our paper is organized as follows. Section 2 contains preliminaries. Section 3 contains problem statements and the proof that our problem is isomorphism complete. The final section considers special cases and draws conclusions.

2 Preliminaries

Let $\mathcal{T}(\mathcal{V}, \mathcal{F})$ be a collection of terms constructed from denumerable sets of variables $\mathcal{V}$ and operators $\mathcal{F}$. Functions in $\mathcal{F}$ include *binding operators* which are instances of *binding operator families*. Specifically a binding operator family represents an denumerable set of binding operators indexed by members of $\mathcal{V}$. For example if our language is that of the λ-calculus, we have a binding operator family λ with instances λ_x, λ_y, ... and $\lambda x.x$ would be short hand for $\lambda_x(x)$.

We call the index variable of an operator θ_x the variable *bound* by θ. (As a notational convention we shall name functions by Greek letters when they are known to bind variables, and use lower case Roman letters otherwise.) We define free and bound variables in the standard way and define substitution to respect variable binding (e.g., see [8] for an account of this in the predicate calculus). For example, if s is a term, the result of substituting t in s for a variable x, denoted by $s[t/x]$ is defined recursively by the following cases:

- $y[t/x] =$ if $y \neq x$ then y else t;
- $f(t_1 ... t_n)[t/x] = f(t_1[t/x], ..., t_n[t/x])$, when f is not a binding operator.
- $\theta_y(s')[t/x] =$ if $x \neq y$ then $\theta_y(s'[t/x])$ else $\theta_y(s')$

In the last condition, to avoid free-variable capture, we assume that t does not contain y free. When this is not the case we must rename bound variables (see α below) to avoid capture. We write $[t_1, t_2, ..., t_n/x_1, x_2, ..., x_n]$ to represent the iterative application of the substitutions $[t_i/x_i]$.

We will be reasoning about the equality of terms where their operators satisfy certain equational axioms. A binary operator f is *associative* when it satisfies

$$f(x, f(y, z)) = f(f(x, y), z) \tag{A}$$

for all x and y, and f is *commutative* when

$$f(x, y) = f(y, x). \tag{C}$$

We call a binding operator family θ commutative when for every $x, y \in \mathcal{V}$ and $t \in \mathcal{T}(\mathcal{V}, \mathcal{F})$

$$\theta_x(\theta_y(t)) = \theta_y(\theta_x(t)). \qquad (C').$$

We call functions that obey both (A) and (C) *AC-functions* and operators that obey (C') *C-binding operators*. Finally, we identify as equal so called α-convertible terms, which are those terms differing only in the names of their bound variables. That is, for every binding operator family $\theta \in \mathcal{F}$, $s \in \mathcal{T}(\mathcal{V}, \mathcal{F})$ and $x \in \mathcal{V}$

$$\theta_x(s) = \theta_y(s[y/x]) \quad \text{where } y \text{ is neither free nor bound in } s \tag{α}$$

We call two terms s and t *associative-commutative equivalent* (denoted by $s \stackrel{\text{ace}}{=} t$) when $s = t$ is an equational consequence can be proven by using instances of (A), (C), (C') (corresponding to functions that are associative and/or commutative) and (α) as axiom schemata. For example, the terms given in (1) and (2) are equal when conjunction satisfies (A) and (C) and existential quantification satisfies (C').

3 Problem Statements and Equivalence Proof

Our proof that deciding AC-term equality is polynomially equivalent to Graph Isomorphism uses the following problems.

1. **(Directed) Graph Isomorphism** (called DGI in the directed case and GI in the undirected case)

   Instance: Two (directed) graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

   Question: Is $G_1 \sim G_2$? That is, is there a bijection $\phi : V_1 \to V_2$ such that $(v_i, v_j) \in E_1$ iff $(\phi(v_i), \phi(v_j)) \in E_2$?

2. **Labeled Directed Graph Isomorphism** (called LDGI)

   Instance: Two directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ that may have labels associated with their vertices and edges.

   Question: Is $G_1 \sim G_2$? That is, is there a bijection $\phi : V_1 \to V_2$ such that $(v_i, v_j) \in E_1$ iff $(\phi(v_i), \phi(v_j)) \in E_2$ and ϕ preserves edge and vertex labelings?

3. **AC-Equality of Terms Containing Commutative Binding Operators.** (called ACE)

   Instance: Two terms $s,t \in \mathcal{T}(\mathcal{V}, \mathcal{F})$ where $\mathcal{F}$ possibly contains associative and commutative functions and commutative binding operators.

   Question: Is $s \stackrel{\text{ace}}{=} t$?

Our equivalence proof proceeds in two parts. First we demonstrate that GI polynomially reduces to ACE.

Lemma 1 *$GI \leq^p ACE$*

Proof: Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be a given instance of GI. Let f be an AC-function, g a commutative function, and θ a commutative binding operator family. We show how to construct terms $s, t \in \mathcal{T}(\mathcal{V}, \mathcal{F})$, where $V_1 \cup V_2 \subset \mathcal{V}$ and $\mathcal{F} = \{f, g, \theta_{v_o}, \theta_{v_1}, ...\})$, such that $s \stackrel{\text{ace}}{=} t$ iff G_1 is isomorphic to G_2. Let $m = |V_1|$ and $n = |E_1|$. Construct s by outermost binding the variables in V_1 with m nested applications of θ_{v_1} through θ_{v_m}. The argument to the final θ_{v_m} is the nested right-associated applications of f to the n occurrences of g, which are in turn applied to the endpoint vertices of the edges in E_1. That is, if the kth edge in E_1 is (v_{i_k}, v_{j_k}) (ordered arbitrarily[1]), then the kth application of g in s will be $g(v_{i_k}, v_{j_k})$. Hence, s is

$$\theta_{v_1}(\dots \theta_{v_m}(f(g(v_{i_1}, v_{j_1}), f(g(v_{i_2}, v_{j_2}), \dots, f(g(v_{i_{n-1}}, v_{j_{n-1}}), g(v_{i_n}, v_{j_n}))\dots)))) \quad (3)$$

We construct t analogously using G_2. Both constructions may be accomplished in polynomial time (and in log space).

Correctness is easy to establish. If ϕ is an isomorphism from G_1 to G_2, then s contains a subterm $g(v_{i_k}, v_{j_k})$ iff t contains either the subterm $g(\phi(v_{i_k}), \phi(v_{j_k}))$ or $g(\phi(v_{j_k}), \phi(v_{i_k}))$ It follows that $s \stackrel{\text{ace}}{=} t$. Conversely, if $s \stackrel{\text{ace}}{=} t$, then there must be a renaming of binding variables in s, such that the resulting term may be rewritten to t using (A), (C), and (C'). This renaming is an isomorphism from G_1 to G_2. $\square$

To prove the converse of Lemma 1, we first define a function $\mathit{frontier}_f(t)$ which returns a list of subterms of t.

$$\mathit{frontier}_f(t) = \begin{cases} \mathit{frontier}_f(a_1) \mathbin{@} \dots \mathbin{@} \mathit{frontier}_f(a_n) & \text{if } t = f(a_1, \dots, a_n) \\ [t] & \text{otherwise} \end{cases}$$

The operator @ appends its operand lists and $[t]$ represents the singleton list with member t. We use the following facts (e.g., see [2, 10]) about *frontier* in our proof.

Fact 1 If f is associative, $\mathit{frontier}_f(s) = [C_1, \dots, C_k]$, $\mathit{frontier}_f(t) = [D_1, \dots, D_k]$, and for $1 \leq i \leq k$, $C_i \stackrel{\text{ace}}{=} D_i$, then $s \stackrel{\text{ace}}{=} t$.

Fact 2 If f is an AC-function, and s and t are terms such that $\mathit{frontier}_f(s)$ is (pointwise) AC-equivalent to a permutation of $\mathit{frontier}_f(t)$, then $s \stackrel{\text{ace}}{=} t$.

We may now prove:

[1] Edges in G_1 and G_2 are unordered; this corresponds to g being commutative.

Lemma 2 *$ACE \leq^p GI$*

Proof: We actually reduce ACE to LDGI. Proofs that LDGI polynomially reduces to DGI may be found in [4, 7]. A reduction of DGI to GI may be found in [4].

We provide a procedure ρ that for any $t \in \mathcal{T}(\mathcal{V}, \mathcal{F})$ returns a rooted labeled directed acyclic graph (LDAG) G_t which roughly corresponds to its parse tree. The reduction consists of applying this procedure to the two terms that comprise the instance of ACE.

We define ρ by recursion on the structure of terms $t \in \mathcal{T}(\mathcal{V}, \mathcal{F})$.

Case 1: $t \equiv c$, where c is a constant
G_t is a vertex labeled by c.

Case 2: $t \equiv x$, where x is a variable
G_t is a vertex labeled by x.

Case 3: $t \equiv f(t_1, ..., t_n)$

Case 3a: f is neither associative nor commutative
Let G_t be rooted by a new vertex labeled by f with edges pointing to the roots of the G_{t_i}. Label the edge to G_{t_i} with the integer i.

Case 3b: f is commutative and is not associative ($n = 2$)
G_t consists of a new vertex labeled by f, with unlabeled edges pointing to the roots of G_{t_1} and G_{t_2}.

Case 3c: f is associative and is not commutative ($n = 2$)
Let G_t be rooted by a new vertex v labeled by f. If $[C_1, .., C_k] =$ $\mathit{frontier}_f(t)$, then for each i from 1 to k, attach a directed edge from v to the root of G_{C_i} and label this edge by i.

Case 3d: f is both associative and commutative ($n = 2$)
Same as Case 3c, except do not label the edges leaving v.

Case 4: $t \equiv \theta_{x_1}(\theta_{x_2}(... \theta_{x_n}(t')...))$ where the outermost term constructor of t' is not θ

Case 4a: θ is not commutative
Let G_t be rooted by a new vertex v labeled by θ. Add an edge from v to $G_{t'}$. In addition, add edges from v to n new vertices $w_1, w_2, ..., w_n$ and label the ith edge by the integer i. For each w_i, add edges from w_i to the vertices in $G_{t'}$ labeled by x_i, and erase the "x_i"-labels on these vertices.

Case 4b: θ is commutative
Same as Case 4a, except do not label the edges from v to the w_i.

If $s \stackrel{\text{ace}}{=} t$ then some finite application of (A),(C),(C'), and bound variable renaming will rewrite s and t to some term w. It is easy to check that each such rewrite does not, up to isomorphism, alter the image of a term under ρ. Conclude that $G_s \sim G_t$.

The converse is more complicated. Formally, we prove by induction on k that for all terms s and t where $\mathit{height}(G_s) \leq k$ and $G_s \sim G_t$, then $s \stackrel{\text{ace}}{=} t$. The function *height* computes the height of its LDAG argument (i.e., the length of the longest path). We proceed by cases on the outermost operator of s (which must be identical to the outermost operator of t since their images under ρ are isomorphic).

Case 1: $s \equiv c$, where c is a constant
As $G_s \sim G_t$, t must also be c.

Case 2: $s \equiv x$, where x is a variable
As $G_s \sim G_t$, t must also be x.

Case 3: $s \equiv f(s_1, ..., s_n)$ and $t \equiv f(t_1, ..., t_n)$

Case 3a: f is neither associative nor commutative
Then $G_s \sim G_t$ establishes an isomorphism between G_{s_i} and G_{t_i}. By the induction hypothesis, we have that $s_i \stackrel{\text{ace}}{=} t_i$, and it follows (as equality is a congruence relation) that $s \stackrel{\text{ace}}{=} t$.

Case 3b: f is commutative and is not associative
Then $G_s \sim G_t$ implies either $G_{s_1} \sim G_{t_1}$ and $G_{s_2} \sim G_{t_2}$, or alternatively, $G_{s_1} \sim G_{t_2}$ and $G_{s_2} \sim G_{t_1}$. Either case is possible as, unlike Case 3a, the edges leaving the roots f are unlabeled. In the first case, we have by the induction hypothesis that $s_1 \stackrel{\text{ace}}{=} t_1$ and $s_2 \stackrel{\text{ace}}{=} t_2$ from which $s \stackrel{\text{ace}}{=} t$ follows by congruence. In the second case $s_1 \stackrel{\text{ace}}{=} t_2$ and $s_2 \stackrel{\text{ace}}{=} t_1$ and our result follows by congruence and the commutativity of f.

Case 3c: f is associative and is not commutative
G_s is a graph rooted by f with children $G_{C_1}, ..., G_{C_k}$ and $[C_1, ..., C_k]$ $=$ $frontier_f(s)$. Similarly G_t is a graph rooted by f with children $G_{D_1}, ..., G_{D_k}$ where $[D_1, ..., D_k] = frontier_f(t)$. Hence, $G_s \sim G_t$ implies that $G_{C_i} \sim G_{D_i}$ and, by the induction hypothesis, it follows that $C_i \stackrel{\text{ace}}{=} D_i$. By Fact 1, $s \stackrel{\text{ace}}{=} t$.

Case 3d: f is both associative and commutative
Same as Case 3c, except that the isomorphism from G_s to G_t only establishes that some permutation of the $G_{C_1}, ..., G_{C_k}$ is isomorphic to the $G_{D_1}, ..., G_{D_k}$. It follows by the induction hypothesis and Fact 2 that $s \stackrel{\text{ace}}{=} t$.

Case 4: $s \equiv \theta_{x_1}(\theta_{x_2}(\, ... \theta_{x_n}(s')...))$ and $t \equiv \theta_{y_1}(\theta_{y_2}(\, ... \theta_{y_n}(t')...))$

Case 4a: θ is not commutative
Then $G_s \sim G_t$ implies that $G_{\sigma(s')} \sim G_{t'}$ where σ is the substitution $[y_1, ..., y_n / x_1, ..., x_n]$. By the induction hypothesis, $\sigma(s') \stackrel{\text{ace}}{=} t'$, and it follows that $\theta_{y_1}(\, ... \theta_{y_n}(\sigma(s'))...) \stackrel{\text{ace}}{=} \theta_{y_1}(\, ... \theta_{y_n}(t')...)$. By renaming bound variables we conclude $s \stackrel{\text{ace}}{=} t$.

Case 4b: θ is commutative
Same as Case 4a, except that σ is a bijection from $\{x_1, ..., x_n\}$ to $\{y_1, ..., y_n\}$. Permuting the names of binding variables does not effect AC-equality as θ is commutative.

□

Putting the previous two lemmas together, we conclude:

Theorem 1 *ACE is polynomial equivalent to GI*

4 Discussion

We have given fairly direct reductions between graph isomorphism and AC-equality. In both directions, the reductions are efficient enough that a previously coded solution for either problem could be used as a decision procedure for the other.

The reduction from ACE to GI also provides insight into the complexity of AC-term equality and the possibility of polynomial decision procedures for special cases. Observe that if we eliminate binding operators (and hence bound variables), then the labeled DAGs associated with terms would be labeled trees and we could decide equality by extending polynomial-time tree isomorphism algorithms [1] to handle vertex and edge labelings. This confirms our observation in the first section: AC-term equality is tractable in theories without binding variables.

Even when binding variables are present, there are cases where deciding AC-equality is still tractable. For example, as planar graph isomorphism is decidable in polynomial time [11], so is AC-equality when the corresponding term graphs contain no subgraphs contractable to K_5 or $K_{3,3}$. Non-planarity arises when terms share multiple bound variables across multiple subterms. For example, three independent subterms containing the same three bound variables will correspond to a graph containing a subgraph contractable to $K_{3,3}$. For example in hardware verification the terms we encounter are relations that represent devices and the only binding variables are existential quantifiers which represent internal wires and are shared by relations that are "wired together". In this domain, many gate-level devices have small fan-in and fan-out and, as a result, share few bound variables; their corresponding graphs are often planar.

Acknowledgements

This work benefited from discussions with Dexter Kozen, Paliath Narendran, and Alexander Bockmayr.

References

[1] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] Leo Bachmair and David A. Plaisted. Associative path orderings. In *First International Conference on Rewriting Techniques and Applications*, pages 241–254, Dijon, France, 1985.

[3] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. In *First International Conference on Rewriting Techniques and Applications*, pages 417–429, Dijon, France, 1985.

[4] Kellogg S. Booth and Charles J. Colbourn. Problems polynomially equivalent to graph isomorphism. Technical Report CS-77-04, University of Waterloo, 1979.

[5] Albert Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers B. V. (North-Holland), 1987.

[6] Nachum Dershowitz et al. Associative-commutative rewriting. In *Eighty International Joint Conference on Artificial Intelligence*, pages 940–944, Karlsruhe, West Germany, 1983.

[7] M. Fontet. Automorphismes de graphes et planarite. *Asterisque*, pages 73–90, 1976.

[8] Jean H. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[10] Bernhard Gramlich and Jörg Denzinger. Efficient ac-matching using constraint propagation. Technical Report SEKI Report SR-88-15, FB Informatik, Universität Kaiserslautern, 1988.

[11] John Hopcroft and J.K. Wong. A linear time algorithm for isomorphism of planar graphs. In *Proc. 6th Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[12] J.M. Hullot. Associative-commutative pattern matching. In *Proceedings of the IJCAI-79*, pages 406–412, Tokyo, Japan, 1979.

[13] David S. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 9:426–444, 1988.

[14] Deepak Kapur and Paliath Narendran. NP-completeness of the set unification and matching problems. In *8th International Conference On Automated Deduction*, pages 489–495, Oxford, UK, 1986.