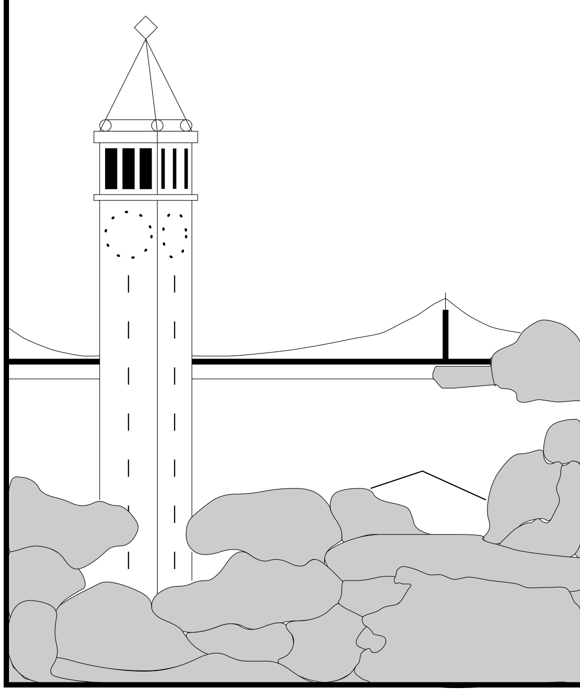
Path-Sensitive Analysis for Linear Arithmetic and Uninterpreted Functions

Sumit Gulwani and George C. Necula



Report No. UCB/CSD-4-1325

May 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Path-Sensitive Analysis for Linear Arithmetic and Uninterpreted Functions

Sumit Gulwani and George C. Necula

University of California, Berkeley
{gulwani,necula}@cs.berkeley.edu

Abstract. We describe data structures and algorithms for performing a path-sensitive program analysis to discover equivalences of expressions involving linear arithmetic or uninterpreted functions. We assume that conditionals are abstracted as boolean variables, which may be repeated to reflect equivalent conditionals. We introduce *free conditional expression diagrams* (FCEDs), which extend binary decision diagrams (BDDs) with internal nodes corresponding to linear arithmetic operators or uninterpreted functions. FCEDs can represent values of expressions in a program involving conditionals and linear arithmetic (or uninterpreted functions). We show how to construct them easily from a program, and give a randomized linear time algorithm (or quadratic time for uninterpreted functions) for comparing FCEDs for equality. FCEDs are compact due to maximal representation sharing for portions of the program with independent conditionals. They inherit from BDDs the precise reasoning about boolean expressions needed to handle dependent conditionals.

1 Introduction

Data structures and algorithms for manipulating boolean expressions (e.g., binary decision diagrams) have played a crucial role in the success of model checking for hardware and software systems. Software programs are often transformed using boolean abstraction [4] to boolean programs: arithmetic operations and other operators are modeled conservatively by their effect on a number of boolean variables that encode predicates on program state. In this paper, we show that we can reason efficiently and precisely about programs that contain not only boolean expressions but also linear arithmetic and uninterpreted functions. Such algorithms are useful when the desired level of precision cannot be achieved with boolean abstraction of linear arithmetic expressions in a program.

Consider the program fragment shown in Figure 1. The atomic boolean expressions in the conditionals (e.g. $x < y, y == z$) have been abstracted as boolean variables c_1 and c_2. We assume that the conditional abstraction procedure can

This research was supported in part by the National Science Foundation Grant CCR-0081588, and gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

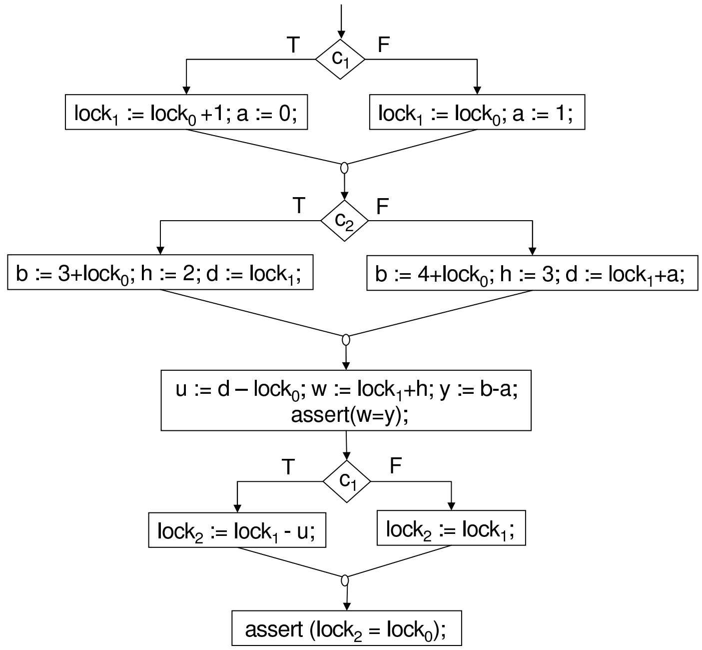


Fig. 1. An example program fragment.

sometimes detect equivalences of atomic boolean expressions (e.g. $x < y$ and $y > x$ are equivalent), as is the case for the first and last conditionals in the program. Suppose our goal is to determine the validity of the two assertions in the program. The first assertion holds because it is established on all four paths that can reach it. The second assertion holds only because the first and last conditionals use identical guards. A good algorithm for verifying these assertions should be able to handle such dependent conditionals (Two conditionals are dependent if truth-value of one depends on the other), or in other words perform a path-sensitive analysis, without individually examining an exponential number of paths that arise for portions of the program with independent conditionals.

Since there is no obvious boolean abstraction for this example, we need to reason about the linear arithmetic directly. There are two kinds of algorithms known to solve this problem. On one extreme, there are abstract/random interpretation based polynomial-time algorithms, which perform a path-insensitive analysis. Karr described a deterministic algorithm [21] based on abstract interpretation [11]. Recently, we gave a faster randomized algorithm [18] based on random interpretation. These algorithms are able to decide the first assertion in the program since the first two conditionals preceding it are independent of each other. However, these algorithms cannot verify that the second assertion holds, because they would attempt to do so over all the eight paths through the program, including four infeasible ones.

On the other extreme, there are multi-terminal binary decision diagram (MTBDD) [15] based algorithms that consider all feasible paths in a program

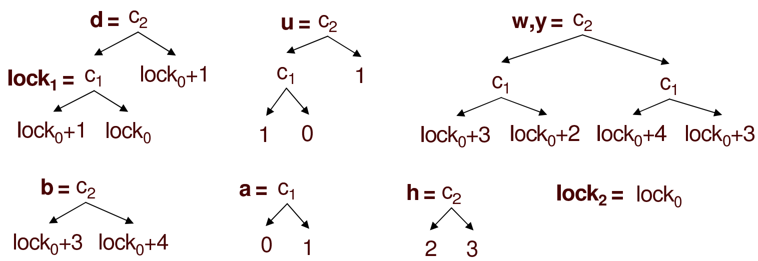


Fig. 2. The MTBDD representation for symbolic values of variables of the program in Figure 1. The internal nodes are conditionals whose left child corresponds to the conditional being true. The leaves are canonicalized linear arithmetic expressions.

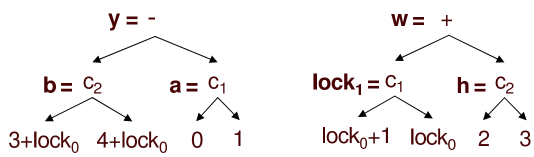


Fig. 3. The VDG/FCED representations for symbolic values of variables of the program in Figure 1. The internal nodes also involve arithmetic operations. This leads to succinct representations, and allows sharing.

explicitly, and hence are able to decide both assertions in our example. However, these algorithms run in exponential time even when most of the conditionals in a program are independent of each other, which is quite often the case. MTBDDs are binary decision diagrams whose leaves are not boolean values but canonicalized linear expressions. For the example program, the MTBDDs corresponding to final values of the various variables are shown in Figure 2. These MTBDDs use the same ordering of boolean variables and the same canonicalization for leaves. With MTBDDs we can verify both assertions; however note that checking equality between w and y essentially involves performing the check individually on each of the four paths from the beginning of the program to the first assertion. Also note that there is little opportunity for sharing subexpressions in a MTBDD due to the need to push computations down to the leaves and to canonicalize the leaves. This algorithm is exponential in the number of boolean variables in the program. Its weak point is the handling of sequences of independent conditionals and its strong point is that it can naturally handle dependent conditionals, just like a BDD does for a boolean program.

In this paper, we describe data structures and algorithms that combine the efficiency of the path-insensitive polynomial-time algorithms with the precision of the MTBDD-based algorithms. Consider representing the values of w and y using value dependency graph (VDG) [27], as shown in Figure 3. Such a representation can be easily obtained by symbolic evaluation of the program. Note that this representation is exponentially more succinct than MTBDDs. For example, note that $|\text{VDG}(y)| = |\text{VDG}(b)| + |\text{VDG}(a)|$ while $|\text{MTBDD}(y)| = |\text{MTBDD}(b)| \times |\text{MTBDD}(a)|$ (here $|\text{VDG}(y)|$ denotes the size of VDG representation for y). This is because VDGs do not need to maintain a normal form for expressions unlike MTBDDs, which even require a normal form for their leaves. For example, w and y, which are equivalent expressions, have distinct VDG representations as shown in Figure 3. A VDG for any expression can share nodes with the VDGs for its subexpressions. For example, note that VDG(y) shares nodes with VDG(b) and VDG(a). On the other hand, an MTBDD typically cannot exploit any sharing that is induced by the order in which a program computes expressions.

The challenge now is to check equivalence of two VDGs. We do not know of any efficient deterministic algorithm to solve this problem. We show in this paper a randomized algorithm that can check equivalence of two *free* VDGs in linear time. A VDG is said to be *free* if every boolean variable occurs at most once on any path from the root node to a leaf. Note that if all conditionals in a program are independent of each other, then the VDG for any expression in the program is free. For example, the VDGs shown in Figure 3 are free.

In this paper, we propose *Free Conditional Expression Diagrams* (FCEDs), which are a generalization of free VDGs. We describe a transformation that generates an FCED for any expression in a loop-free program, and a randomized algorithm that checks equivalence of two FCEDs in linear time. This, in turn, gives an algorithm for checking the validity of assertions $e_1 = e_2$ in programs that contain linear arithmetic and conditionals. This algorithm is more efficient than the MTBDD-based algorithm. In particular, if all conditionals in a program are independent of each other, then this algorithm is as fast as the random interpretation based algorithm, which runs in polynomial time, as opposed to the MTBDD-based algorithm, which has exponential cost. However, the new algorithm still has the same worst-case complexity as the MTBDD-based algorithm (This happens when all conditionals in the program are arbitrary boolean expressions involving the same set of boolean variables). This is not surprising since the problem of checking equality assertions in a program with dependent conditionals is NP-hard and it is generally believed that even randomized algorithms cannot solve such problems in polynomial time.

In Section 2, we describe the FCED construction and the randomized equivalence testing algorithm for conditional linear arithmetic expressions. In Section 3, we describe the FCED construction and the randomized equivalence testing algorithm for conditional uninterpreted function terms.

2 Analysis for Linear Arithmetic

2.1 Problem Definition

Let $\mathcal{L}_a$ be the following conditional arithmetic expression language over rational constants q, rational variables x, boolean variables c, and boolean expressions b.

$$e ::= q \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid q \times e \mid \textit{if } b \textit{ then } e_1 \textit{ else } e_2$$
$$b ::= c \mid b_1 \wedge b_2 \mid b_1 \vee b_2$$

We want a data structure FCED to succinctly represent the expressions in language $\mathcal{L}_a$ and support efficient algorithms for the following two problems:

P1. Given the FCEDs for the sub-expressions of an expression $e \in \mathcal{L}_a$, construct the FCED for the expression e.
P2. Given the FCED representations for two expressions $e_1, e_2 \in \mathcal{L}_a$, decide whether $e_1 = e_2$.

Note that the symbolic value of any expression in our example program belongs to the language $\mathcal{L}_a$. For example, the value of $lock_1$ is " *if* c_1 *then* $lock_0 + 1$ *else* $lock_0$". Hence, algorithms for problems P1 and P2 can be used to check equivalence of two expressions in a loop-free program. In general, if a program has loops, then since the lattice of linear equality facts has finite height k (where k is the number of variables in the program), one can analyze a suitable unrolling of the loops in the program to verify the assertions [21, 18].

Note that we assume that there is an abstraction procedure for conditionals that maps atomic conditionals to boolean variables such that only equivalent conditionals are mapped to the same boolean variable. Equivalent conditionals can be detected by using standard value numbering heuristics [24, 1] (e_1 *relop* $e_2 \equiv e_1'$ *relop* e_2' if $e_1 = e_1'$ and $e_2 = e_2'$ and *relop* $=$ *relop'*) or other sophisticated heuristics [23] (e.g. e_1 *relop* $e_2 \equiv e_1'$ *relop'* e_2' if $e_1 - e_2 = e_1' - e_2'$ and *relop* $=$ *relop'*). Here *relop* stands for a relational operator, e.g. $=, <$ or $>$. Note that detecting equivalence of conditionals involves detecting equivalence of expressions, which in turn can be done by using a simple technique like value numbering. We can even use the result of our analysis to detect those equivalences on the fly.

2.2 FCED Construction

An FCED for linear arithmetic is a DAG generated by the following language over rational constants q, rational variables x and boolean expressions g, which we call guards.

$$\begin{aligned} f ::= x \;\mid\; q \;\mid\; Plus(f_1, f_2) \;\mid\; Minus(f_1, f_2) \\ \mid\; Times(q, f) \;\mid\; Choose(f_1, f_2) \;\mid\; Guard_g(f) \end{aligned}$$

The *Choose* and *Guard* node types are inspired by Dijkstra's guarded command language [14]. Given a boolean assignment ρ, the meaning of $Guard_g(f)$ is either the meaning of f (if g is true in ρ) or undefined (otherwise). The meaning of a *Choose* node is the meaning of its child that is defined. The *Choose* operator here is deterministic in the sense that at most one of its children is defined given any boolean assignment.

The guards g are represented using Reduced Ordered Binary Decision Diagrams (ROBDDs). Let $\preccurlyeq$ be the total ordering on program variables used in these ROBDD representations. For any sets of boolean variables B_1 and B_2, we use the notation $B_1 \prec B_2$ to denote that $B_1 \cap B_2 = \emptyset$ and $c_1 \preccurlyeq c_2$ for all variables

$c_1 \in B_1$ and $c_2 \in B_2$. The guards g can be described by the following language over boolean variables c.

$$g ::= true \quad | \quad false \quad | \quad c \quad | \quad If(c, g_1, g_2)$$

We assume that we can compute conjunction ($\wedge$) of two guards and negation ($\neg$) of a guard. For any boolean guard g, let $BV(g)$ denote the set of boolean variables that occur in g. Similarly, for any FCED node f, let $BV(f)$ denote the set of boolean variables that occur below node f. An FCED f must satisfy the following invariant:

Invariant 1 *For any guard node $Guard_{g_1}(f_1)$ in FCED f, $BV(g_1) \prec BV(f_1)$.*

Invariant 1 is similar to the ROBDDs' requirement that boolean variables on any path from the root node to a leaf must be ordered. As we shall see, it plays an important role in the randomized equivalence testing algorithm that we propose.

The FCED representation of any expression e is denoted by $FCED(e)$ and is computed inductively as follows:

$$\begin{aligned}
FCED(x) &= x \\
FCED(q) &= q \\
FCED(e_1 + e_2) &= Plus(FCED(e_1), FCED(e_2)) \\
FCED(e_1 - e_2) &= Minus(FCED(e_1), FCED(e_2)) \\
FCED(q \times e) &= Times(q, FCED(e)) \\
FCED(\textit{if } b \textit{ then } e_1 \textit{ else } e_2) &= Choose(||g_b, FCED(e_1)||, ||\neg g_b, FCED(e_2)||),
\end{aligned}$$

where g_b is the ROBDD representation of the boolean expression b as a guard. The normalization operator $||g, f||$ takes as input a boolean guard g and an FCED f and returns another FCED whose meaning is equivalent to $Guard_g(f)$, except that Invariant 1 is satisfied:

$$\begin{aligned}
||g, f|| &= Guard_g(f), \quad \textit{if } BV(g) \prec BV(f) \\
||g, f|| &= Guard_g(f[g]), \quad \textit{if } g \textit{ is a conjunction of literals} \\
||g, Plus(f_1, f_2)|| &= Plus(||g, f_1||, ||g, f_2||) \\
||g, Minus(f_1, f_2)|| &= Minus(||g, f_1||, ||g, f_2||) \\
||g, Times(q, f')|| &= Times(q, ||g, f'||) \\
||g, Choose(f_1, f_2)|| &= Choose(||g, f_1||, ||g, f_2||) \\
||g, Guard_{g'}(f')|| &= Guard_{g'}(||g, f'||), \quad \textit{if } BV(g') \prec BV(g) \\
&= ||g \wedge g', f'|| \quad \textit{otherwise}
\end{aligned}$$

where $f[g]$ denotes the FCED obtained from f by replacing any boolean variable c by true or false, if it occurs in g in non-negated or negated form respectively. The purpose of the normalization $||g, f||$ is to simplify f or to push the guard g down into f until a point when the boolean variables in g and f are disjoint, thus ensuring that Invariant 1 is maintained. Figure 4 shows the FCED for variable

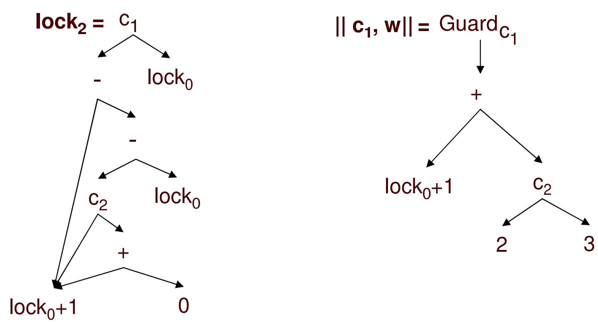


Fig. 4. An example of FCED and normalization operator

$lock_2$ in our example program. Figure 4 also shows the FCED for $||c_1, w||$, where the FCED for w has been shown in Figure 3. We use the notation $c(f_1, f_2)$ as a syntactic sugar for the FCED $Choose(Guard_c(f_1), Guard_{\neg c}(f_2))$. We also simplify an FCED $Choose(Guard_g(f_1), Guard_{false}(f_2))$ to $Guard_g(f_1)$.

2.3 Randomized Equivalence Testing

In this section, we describe an algorithm that decides equivalence of two FCEDs. The algorithm assigns a hash value $V(n)$ to each node n in an FCED, computed in a bottom-up manner from the hash values of its immediate children. The hash value of an FCED is defined to be the hash value assigned to its root. Two FCEDs are declared equivalent iff they have same hash values. This algorithm has a one-sided error probability. If two FCEDs have different hash values, then they are guaranteed to be non-equivalent. However, if two FCEDs are not equivalent, then there is a very small probability (over the random choices made by the algorithm) that they will be assigned same hash values. The error probability can be made arbitrarily small by setting the parameters of the algorithm appropriately.

For the purpose of assigning a hash value to an FCED representation of any expression in $\mathcal{L}_a$, we choose a random value for each of the boolean and rational variables. The random values for both kind of variables are chosen independently of each other and uniformly at random from some finite set of rationals. (Note that we choose a rational random value even for boolean variables). For any variable y, let r_y denote the random value chosen for y. The hash value $V(n)$ is assigned inductively to any node n in an FCED as follows:

$$\begin{aligned} V(q) &= q \\ V(x) &= r_x \\ V(Plus(f_1, f_2)) &= V(f_1) + V(f_2) \\ V(Minus(f_1, f_2)) &= V(f_1) - V(f_2) \\ V(Times(q, f)) &= q \times V(f) \\ V(Choose(f_1, f_2)) &= V(f_1) + V(f_2) \\ V(Guard_g(f)) &= H(g) \times V(f) \end{aligned}$$

where the hash function H for a boolean guard g is as defined below.

$$\begin{aligned} H(true) &= 1 \\ H(false) &= 0 \\ H(c) &= r_c \\ H(If(c, g_1, g_2)) &= r_c \times H(g_1) + (1 - r_c) \times H(g_2) \end{aligned}$$

For example, note that $w = (\textit{if } c_1 \textit{ then } lock_0{+}1 \textit{ else } lock_0){+}(\textit{if } c_2 \textit{ then } 2 \textit{ else } 3)$ and $y = (\textit{if } c_2 \textit{ then } 3{+}lock_0 \textit{ else } 4{+}lock_0){-}(\textit{if } c_1 \textit{ then } 0 \textit{ else } 1)$ in our example program. If we choose $r_{lock_0} = 3, r_{c_1} = 5, r_{c_2} = -3$, then $V(w) = V(y) = 14$, thereby validating the assertion $w = y$. If we choose random boolean values for boolean variables while computing hash values, then we would essentially be hashing the symbolic values of expressions on one random path (corresponding to the random boolean choice). However, it is essential to check for the equivalence of expressions on *all* paths. Choosing non-boolean random values for boolean variables help us to do that by essentially computing a random weighted combination of the hash values of expressions on all paths. In the next section, we explain more formally why, with high probability, this hashing scheme assigns equal values only to equivalent expressions.

2.4 Completeness and Probabilistic Soundness of the Algorithm

Let e be any expression in language $\mathcal{L}_a$. Let $P(FCED(e))$ denote the polynomial obtained by using variables x and c instead of random values r_x and r_c, while computing $V(FCED(e))$. The following properties hold.

T1. $V(FCED(e))$ is the result of evaluating the polynomial $P(FCED(e))$ at random values r_y chosen for each variable y that occurs in $P(FCED(e))$.
T2. For any FCED f, $P(f)$ is a multi-linear polynomial, i.e. the degree of any variable is at most 1. This is due to the freeness property of an FCED (ensured by Invariant 1).
T3. $e_1 = e_2$ iff $P(FCED(e_1))$ and $P(FCED(e_2))$ are equivalent polynomials.

Property T1 is trivial to prove. The proof of property T2 is based on the observation that $H(g)$ is multi-linear for any guard g (this is because every boolean variable occurs at most once on any path from the root node to a leaf in an ROBDD), and for any $Guard_g(f)$ node in an FCED, $BV(g) \cap BV(f) = \emptyset$. The proof of property T3 is given in Appendix A.

These properties imply that the equivalence testing algorithm is complete, i.e., it assigns same hash values to equal expressions. Suppose e_1 and e_2 are equal expressions. It follows from T3 that $P(FCED(e_1)) = P(FCED(e_2))$. Since $P(FCED(e_1))$ and $P(FCED(e_2))$ are multi-linear (implied by T2), they are equivalent even when the boolean variables are treated as rational variables. This is a standard fact and is the basis of several algorithms [5, 17, 13, 12]. Therefore, it follows from T1 that $V(FCED(e_1)) = V(FCED(e_2))$.

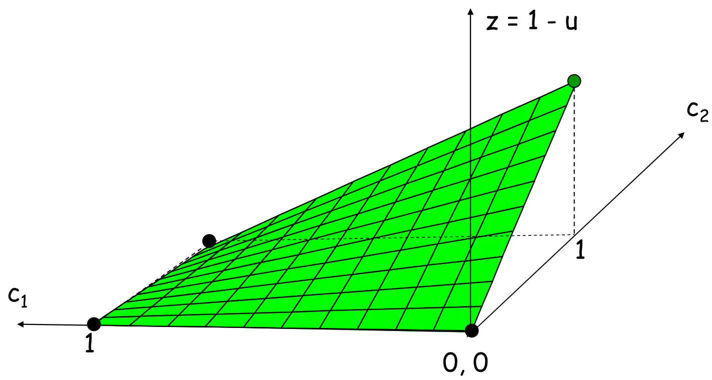


Fig. 5. The surface shows values of expression $1 - u$ for different values of c_1 and c_2.

Properties T1 and T3 imply that the algorithm is probabilistically sound, i.e., it assigns different hash values to non-equivalent expressions with high probability over the random choices that it makes. Suppose $e_1 \neq e_2$. It follows from T3 that $P(FCED(e_1)) \neq P(FCED(e_2))$. Trivially, $P(FCED(e_1)) \neq P(FCED(e_2))$ even when boolean variables are treated as rational variables. It then follows from the classic Schwartz's theorem [26] (on testing equivalence of two polynomials) that the probability that $P(FCED(e_1))$ and $P(FCED(e_2))$ evaluate to the same value on random assignment is bounded above by $\frac{d}{s}$, where d is the maximum of the degrees of the polynomials $P(FCED(e_1))$ and $P(FCED(e_2))$ (these are bounded above by the size of the expressions e_1 and e_2 respectively), and s is the size of the set from which random values are chosen. Therefore, it follows from T1 that $Pr[V(FCED(e_1)) \neq V(FCED(e_2))] \geq 1 - \frac{d}{s}$. (Here $Pr[V(FCED(e_1)) \neq V(FCED(e_2))]$ denotes the probability of the event $V(FCED(e_1) \neq V(FCED(e_2))$ over the choice of the random values r_y for all variables y.)

Note that the error probability can be made arbitrarily small by choosing random values from a large enough set. For boolean variables, this set cannot contain more than 2 elements. It is precisely for this reason that we require property T2, so as to be able to treat boolean variables as rational variables without affecting equivalences of polynomials. Note that multi-linearity is a necessary requirement. For example, consider the two equivalent polynomials $c_2c_3 + c_1^2$ and $c_1 + c_3c_2$ over the boolean variables c_1 and c_2. These polynomials are not equivalent when the variables c_1 and c_2 are interpreted as rational variables since the first polynomial is not multi-linear in c_1.

This randomized algorithm for equivalence checking can be explained informally using a geometric argument. For example, consider the validity of the statement $u = 1$ at the place of the first assertion in Figure 1. This statement is false since it holds on only three of the four paths that reach it. It is false when c_1 is false and c_2 is true. Figure 5 shows a surface in a 3-dimensional space whose z coordinate reflects the value of expression $1 - u$ as a function of (rational) assignment for c_1 and c_2. Since there is at least one boolean assignment for c_1 and c_2 where $1 - u$ is not zero, and since the degree of the surface is small (2

$e =$	$e_1 \pm e_2$	if b then e_1 else e_2	$q \times e_1$	$F(e_1, e_2)$
$T(e)$ for FCED =	*Constant*	$S(g_b) \times (S(e_1) + S(e_2))$	*Constant*	*Constant*
$T(e)$ for MTBDD =	$S(e_1) \times S(e_2)$	$S(g_b) \times (S(e_1) + S(e_2))$	$S(e_1)$	$S(e_1) \times S(e_2)$

Fig. 6. A table comparing the time and space complexity $T(e)$ for constructing FCEDs and MTBDDs of an expression from the representation of its subexpressions.

in this case), it follows that the surface intersects the c_1-c_2 plane in a "small" number of points. This allows the quick discovery, with high probability, of this false assertion by random sampling of the surface (this corresponds to choosing random rational values for boolean variables). If, on the other hand, the surface corresponds to a true assertion, then it is included in the c_1-c_2 plane and any sampling would verify that.

2.5 Time and Space Complexity

The time required to compute the hash value of an FCED is clearly linear in the size of the FCED. However, this is under the assumption that all basic arithmetic operations (like addition, multiplication) to compute the hash value can be performed in unit time. This assumption is not necessarily true since the size of the numbers involved may increase with each arithmetic operation. The standard technique to deal with this problem is to do the arithmetic operations modulo a randomly chosen prime p [22]. This makes sure that at each stage, the numbers can be represented within a constant number of bits and hence each arithmetic operation can be performed in constant time. The modular arithmetic adds an additional small probability of error in our algorithm.

The time and extra space $T(e)$ required to construct the FCED of an expression e from the FCEDs of the subexpressions of e depends on the structure of e. If e is of the form q, x, $e_1 \pm e_2$, or $q \times e$, then it is easy to see that $T(e)$ is constant. If e is of the form *if* b *then* e_1 *else* e_2, then an amortized cost analysis would show that $T(e) = O(S(g_b) \times (S(e_1) + S(e_2)))$, where $g_b = ROBDD(b)$ and $S(g_b)$ denotes the size of the ROBDD g_b. $S(e_1)$ denotes the size of the FCED of expression e_1 (when represented as a tree; however, the boolean guards in e_1 may be represented as DAGs). The upper bound on time complexity for this case relies on Invariant 1 and assumes some sharing of common portions of ROBDDs that arise while construction of $FCED(e)$.

If all conditionals in a program are independent of each other, then, it is easy to see that $FCED(e)$ is linear in size of e, as opposed to the possibly exponential size implied by the above-mentioned bounds on $T(e)$. Figure 6 compares $T(e)$ for FCED and MTBDD representations. The last column in the table refers to the next section.

3 Analysis for Uninterpreted Functions

Reasoning precisely about program operators other than linear arithmetic operators is in general undecidable. A commonly used abstraction is to model any n-ary non-linear program operator as an uninterpreted function under the theory of equality, which has only one axiom, namely, $F(x_1, .., x_n) = F'(x_1, .., x'_n) \iff F = F'$ and $x_i = x'_i$ for all $1 \le i \le n$. The process of detecting this form of equivalence, where the operators are treated as uninterpreted functions, is also referred to as value numbering. In this section, we describe how to construct FCEDs for uninterpreted functions.

3.1 Problem Definition

Let $\mathcal{L}_u$ be the following language over boolean expressions b, variables x and an uninterpreted function symbol F of arity two.

$$e ::= x \mid F(e_1, e_2) \mid \textit{if } b \textit{ then } e_1 \textit{ else } e_2$$

For simplicity, we consider only one binary uninterpreted function F. Our results can be extended easily to languages with any finite number of uninterpreted functions of any finite arity. However, note that this language does not contain any linear arithmetic operators.

We want a data structure to succinctly represent the expressions in language $\mathcal{L}_u$ and support efficient algorithms for the problems similar to those mentioned in Section 2.1. This would be useful to check equivalence of two expressions in any loop-free program. As before, it turns out the lattice of sets of equivalences among uninterpreted function terms has finite height k (where k is the number of variables in the program). Hence, if a program has loops, then one can analyze a suitable unrolling of loops in the program to verify assertions [19, 20].

3.2 FCED Construction

An FCED in this case is a DAG generated by the following language over variables x and boolean guards g represented using ROBDDs.

$$\begin{aligned} f &::= x \mid F(f_1, f_2) \mid \textit{Choose}(f_1, f_2) \mid \textit{Guard}_g(f) \\ g &::= \textit{true} \mid \textit{false} \mid c \mid If(c, g_1, g_2) \end{aligned}$$

Here c denotes a boolean variable. As before, an FCED satisfies Invariant 1. The FCED representation of any expression e is computed inductively as follows:

$$\begin{aligned} FCED(x) &= x \\ FCED(F(e_1, e_2)) &= F(FCED(e_1), FCED(e_2)) \\ FCED(\textit{if } b \textit{ then } e_1 \textit{ else } e_2) &= \textit{Choose}(||g_b, FCED(e_1)||, ||\neg g_b, FCED(e_2)||) \end{aligned}$$

where $g_b = \textit{ROBDD}(b)$. The normalization operator $||g, f||$ takes as input a boolean guard g and an FCED f and returns another FCED as follows:

$$\begin{aligned} ||g, f|| &= \textit{Guard}_g(f), \quad \textit{if } BV(g) \prec BV(f) \\ ||g, f|| &= \textit{Guard}_g(f[g]), \quad \textit{if } g \textit{ is a conjunction of literals} \\ ||g, F(f_1, f_2)|| &= F(||g, f_1||, ||g, f_2||) \\ ||g, \textit{Choose}(f_1, f_2)|| &= \textit{Choose}(||g, f_1||, ||g, f_2||) \\ ||g, \textit{Guard}_{g'}(f')|| &= \textit{Guard}_{g'}(||g, f'||), \quad \textit{if } BV(g') \prec BV(g) \\ &= ||g \wedge g', f'|| \quad \textit{otherwise} \end{aligned}$$

where $f[g]$, $BV(g)$ and $BV(f)$ are as defined before in Section 2.2.

3.3 Randomized Equivalence Testing

The hash values assigned to nodes of FCEDs of expressions in the language $\mathcal{L}_u$ are vectors of k rationals, where k is the largest depth of any expression that arises. We define $\text{depth}(x) = 1$, $\text{depth}(F(e_1, e_2)) = 1+max(\text{depth}(e_1), \text{depth}(e_2))$. For the purpose of assigning hash values, we choose a random value r_y for each variable y and two random $k \times k$ matrices M and N. The following entries of the matrices M and N are chosen independently of each other and uniformly at random from some set of rationals: $M_{i-1,i}, N_{i-1,i}$ for all $2 \leq i \leq k$. The rest of the entries are chosen to be 0. The hash value $V(n)$ is assigned inductively to any node n in an FCED as follows:

$$\begin{aligned} V(x) &= [r_x, \ldots, r_x] \\ V(F(f_1, f_2)) &= V(f_1) \times M + V(f_2) \times N \\ V(Choose(f_1, f_2)) &= V(f_1) + V(f_2) \\ V(Guard_g(f)) &= H(g) \times V(f) \end{aligned}$$

where $H(g)$ is as defined before in Section 2.3. Note that $H(g) \times V(f)$ denotes multiplication of vector $V(f)$ by the scalar $H(g)$, while $V(f_1) \times M$ denotes multiplication of vector $V(f_1)$ by the matrix M.

The proof of property T3 (given in Appendix B) explains the reason behind this fancy hashing scheme. Here is some informal intuition. To maintain multi-linearity, it is important to choose a random linear interpretation for the uninterpreted function F. However, if we let $k = 1$, the hashing scheme cannot always distinguish between non-equivalent expressions. For example, consider $e_1 = F(F(x_1, x_2), F(x_3, x_4))$ and $e_2 = F(F(x_1, x_3), F(x_2, x_4))$. Note that $e_1 \neq e_2$ but $V(FCED(e_1)) = V(FCED(e_2)) = r_{x_1}m^2 + r_{x_2}mn + r_{x_3}nm + r_{x_4}n^2$, where m and n are some random rationals. This happens because scalar multiplication is commutative. This problem is avoided if we work with vectors and matrices because matrix multiplication is not commutative.

3.4 Completeness and Probabilistic Soundness of the Algorithm

Let e be any expression in language $\mathcal{L}_u$. Let $P(FCED(e))$ denote the k^{th} polynomial in the symbolic vector obtained by using variable names x and c instead of random values r_x and r_c, and by using variable names $M_{i,j}$ and $N_{i,j}$ instead of random values for the matrix entries, while computing $V(FCED(e))$. The properties T1,T2,T3 stated in Section 2.4 hold here also. Properties T1 and T2 are easy to prove as before. However, the proof of property T3 is non-trivial, and is given in Appendix B. These properties imply that the randomized equivalence testing algorithm is complete and probabilistically sound as before. The error probability is bounded above by $\frac{d}{s}$, where d and s are as mentioned in Section 2.4.

3.5 Time and Space Complexity

The time required to compute the hash value for an FCED f is $O(n \times k)$ where n is the size of f and k is the size of the largest FCED in the context. The time and extra space $T(e)$ required to construct FCED of an expression e in language $\mathcal{L}_u$ from the FCED of its sub-expressions can be estimated similarly as in Section 2.5, and is shown in Figure 6.

4 Comparison with Related Work

Path-insensitive version of the analyses that we have described in this paper have been well studied. Karr described a polynomial-time abstract interpretation based algorithm [21] to reason precisely about linear equalities in a program with non-deterministic conditionals. Recently, we described a more efficient algorithm based on the idea of random interpretation [18]. Several polynomial-time algorithms have been described in literature for value numbering, which is the problem of discovering equivalences among program expressions when program operators are treated as uninterpreted [1, 25]. All these algorithms are complete for basic blocks, but are imprecise in the presence of joins and loops in a program. Recently, we described algorithms for global value numbering that discover all equivalences among expressions under the assumption that all conditionals are non-deterministic and program operators are uninterpreted [19, 20].

Karthik Gargi described a path-sensitive global value numbering algorithm [16] that first discovers equivalent conditionals, and then uses that information to do a simple predicated global value numbering. However, this algorithm is not complete and cannot handle conditionals as precisely as our algorithm. Our algorithm is complete with respect to the abstraction of conditionals to boolean variables. Gargi's algorithm treats all operators as uninterpreted and hence does not handle linear arithmetic.

The model checking community has been more concerned with path-sensitivity, in an attempt to do whole state-space exploration. The success of ROBDDs has inspired efforts to improve their efficiency and to expand their range of applicability [7]. Several generalizations of ROBDDs have been proposed for efficient boolean manipulation [2, 17]. There have been some efforts to extend the concept to represent functions over boolean variables that have non-boolean ranges, such as integers or real numbers (e.g.Multi Terminal Binary Decision Diagrams (MTBDDs) [3, 9], Edge-Valued Binary Decision Diagrams (EVBDDs), Binary Moment Diagrams (BMDs) [6] and Hybrid Decision Diagrams (HDDs) [8]). Multiway Decision Graphs (MDGs) have been proposed to represent quantifier-free formulas over terms involving function symbols [10]. None of the above mentioned extensions and generalizations of ROBDDs seem well-suited for software model checking since they do not directly and efficiently support manipulation of conditional expressions, i.e. expressions that are built from boolean expressions and expressions from some other theory like that of arithmetic or uninterpreted functions. This is because most of these techniques rely on having a canonical representation for expressions. Figure 2 illustrates the problems that arise with

	Handles Dependent Conditionals	Handles Arithmetic	Handles Independent Conditionals	Randomized or Deterministic
ROBDD	good	no	good	deterministic
MTBDD	good	poor	poor	deterministic
FBG	no	no	good	randomized
RI	no	good	good	randomized
FCED	good	good	good	randomized

Fig. 7. A table comparing different data structures for software model-checking

canonicalization. However, our proposed representation, FCED, can efficiently represent and manipulate such expressions since it does not require a canonical representation.

The idea behind hashing boolean guards g in our randomized equivalence testing algorithm is similar to that used for checking equivalence of Free Boolean Graphs (FBG) [5], FBDDs [17] and d-DNNFs [13, 12] all of which represent boolean expressions. We have extended this line of work with checking equivalence of conditional arithmetic expressions or conditional expressions built from uninterpreted function terms. Similar ideas have also been used in the random interpretation (RI) technique for linear arithmetic [18] and for uninterpreted function terms [19] for detecting equivalence of conditional expressions that involve independent conditionals. Figure 7 compares these related techniques.

5 Conclusion and Future Work

We describe in this paper a compact representation of expressions involving conditionals and linear arithmetic (or uninterpreted functions) such that they can be compared for equality in an efficient way. In the absence of linear arithmetic and uninterpreted functions, our technique behaves like ROBDDs. In fact, FCEDs inherit from ROBDDs the precise handling of dependent conditionals necessary for discriminating the feasible paths in a program with dependent conditionals. However, the main strength of FCEDs is the handling of the portions of the program with independent conditionals. In those situations, the size of FCEDs and the time to compare two FCEDs is linear (quadratic for uninterpreted functions) in the size of the program.

The simpler problem involving only independent conditionals can be solved in polynomial time by deterministic [21, 20] and randomized algorithms [18, 19]. In this special case, randomization brings a lower computational complexity and the simplicity of an interpreter, without having to manipulate symbolic data structures. Once we allow dependent conditionals, the problem becomes NP-hard and we should not expect randomization alone to solve it in polynomial time. We show in this paper that randomization can still help even for NP-hard problems, if we combine it with a symbolic algorithm. We expect that there are other NP-hard program analysis problems that can benefit from integrating the symbolic techniques with randomization.

The next step is to implement our algorithms and compare them with the existing algorithms with regard to running time and number of equivalences discovered. The results of our algorithm can also be used as a benchmark to measure the number of equivalences that are missed by path-insensitive algorithms.

We have presented randomized algorithms for checking equivalence of two FCEDs for the languages $\mathcal{L}_u$ and $\mathcal{L}_a$. It is an open problem to extend these results to the combined language, i.e. the language that involves both conditional arithmetic expressions as well as conditional uninterpreted function terms. It would also be useful to extend these results to other languages/theories apart from linear arithmetic and uninterpreted functions, for example, the theory of lists, the theory of uninterpreted functions modulo commutativity, associativity, or both. Such theories can be used to model program operators more precisely.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on POPL*, pages 1–11. ACM, 1988.
2. H. Andersen and H. Hulgaard. Boolean expression diagrams. In *12th Annual IEEE Symposium on Logic in Computer Science*, pages 88–98. IEEE, June 1997.
3. R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191. ACM/IEEE, Nov. 1993.
4. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '00 Conference on PLDI.* ACM, May 2001.
5. M. Blum, A. Chandra, and M. Wegman. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10:80–82, 1980.
6. R. Bryant and Y. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *32nd ACM/IEEE Design Automation Conference*, June 1995.
7. R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer Aided Design*, pages 236–245. IEEE Computer Society Press, Nov. 1995.
8. E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *International Conference on Computer Aided Design*, pages 159–163. IEEE Computer Society Press, Nov. 1995.
9. E. M. Clarke, K. L. McMillan, X. Znao, M. Fujiia, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 54–60, June 1993.
10. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design: An International Journal*, 10(1):7–46, Feb. 1997.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.
12. A. Darwiche. A compiler for deterministic decomposable negation normal form. In *Proceedings of the Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pages 627–634. AAAI Press, July 2002.

13. A. Darwiche and J. Huang. Testing equivalence probabilistically. Technical Report D-23, Computer Science Department, UCLA, June 2002.
14. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.
15. M. Fujita and P. C. McGeer. Introduction to the special issue on multi-terminal binary decision diagrams. *Formal Methods in System Design*, 10(2/3), Apr. 1997.
16. K. Gargi. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, volume 37, 5, pages 45–56. ACM Press, June 17–19 2002.
17. J. Gergov and C. Meinel. Efficient boolean manipulation with OBDDs can be extended to FBDDs. *IEEE Trans. on Computers*, 43(10):1197–1209, Oct. 1994.
18. S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *30th Annual ACM Symposium on POPL*. ACM, Jan. 2003.
19. S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *31st Annual ACM Symposium on POPL*. ACM, Jan. 2004.
20. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis Symposium*, LNCS. Springer, 2004.
21. M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.
22. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
23. G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN '00 Conference on PLDI*, pages 83–94. ACM, jun 2000.
24. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM, 1988.
25. O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Static Analysis Symposium*, volume 1694 of *LNCS*, pages 232–247. Springer, 1999.
26. J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701–717, Oct. 1980.
27. D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: representation without taxation. In *21st Annual ACM Symposium on POPL*. ACM, Jan. 1994.

A Proof of Property T3 for Language $\mathcal{L}_a$

We first introduce some notation. Let g be any boolean guard. Let $P(g)$ denote the polynomial obtained by using variable names c instead of random values r_c while computing $H(g)$. For any boolean assignment ρ and any polynomial P, we use the notation $P[\rho]$ to refer to the polynomial $P[c_1 = toInt(\rho(c_1)), \ldots, c_n = toInt(\rho(c_n))]$, where $c_1, \ldots, c_n$ are the boolean variables that occur in polynomial P, and $toInt(true) = 1$ and $toInt(false) = 0$. In other words, the polynomial $P[\rho]$ refers to the polynomial obtained from P by substituting all boolean variables c in P by 1 or 0 depending upon whether $\rho(c) = true$ or $\rho(c) = false$ respectively. For any boolean assignment ρ and any expression e, we use the notation $[\![e]\!]\rho$ to refer to the expression obtained from e by partially evaluating it under the boolean assignment ρ. Note that $[\![e]\!]\rho$ does not contain any if constructor. We use the notation $[\![g]\!]\rho$ to refer to the evaluation of boolean guard g under the boolean assignment ρ. Note that $[\![g]\!]\rho$ is either $true$ or $false$.

We now state some useful lemmas.

Lemma 1. *For any boolean assignment ρ and any boolean guard g, we have*

$$\begin{aligned} P(g)[\rho] &= 1, \ \textit{if } [\![g]\!]\rho = true \\ &= 0, \ \textit{otherwise} \end{aligned}$$

Lemma 1 can be proved by induction on structure of g.

Lemma 2. *For any boolean assignment ρ and any FCED f, we have*

$$\begin{aligned} P(||g, f||)[\rho] &= P(f)[\rho], \ \textit{if } [\![g]\!]\rho = true \\ &= 0, \qquad\qquad \textit{otherwise} \end{aligned}$$

Lemma 2 can be proved by induction on structure of FCED f using Lemma 1.

We now state the main theorem.

Theorem 1. *For any boolean assignment ρ and any expression $e \in \mathcal{L}_a$, we have*

$$[\![e]\!]\rho = P(FCED(e))[\rho]$$

Theorem 1 can be proved by induction on structure of expression e using Lemma 2. The proof of property T3 follows easily from this theorem.

B Proof of Property T3 for Language $\mathcal{L}_u$

We first introduce some notation. For any integer $i \in \{1, .., k\}$, let $P(FCED(e), i)$ denote the denote the i^{th} polynomial in the symbolic vector obtained by using variable names x and c instead of random values r_x and r_c, and by using variable names $M_{i,j}$ and $N_{i,j}$, instead of random values, for the matrix entries, while computing $V(FCED(e))$. Note that $P(FCED(e))$ as defined earlier in Section 3.4 is equal to $P(FCED(e), k)$. Let $\mathcal{L}_u^s$ be the following language of uninterpreted function terms without conditional expressions.

$$e ::= x \quad | \quad F(e_1, e_2)$$

We now state some useful properties and lemmas.

Property 1. Let $e_1, e_2, e_1', e_2' \in \mathcal{L}_u^s$. For any integer $i > 1$, $P(FCED(F(e_1, e_2)), i) = P(FCED(F(e_1', e_2')), i)$ iff

(a) $P(FCED(e_1), i-1) \;=\; P(FCED(e_1'), i-1)$, and
(b) $P(FCED(e_2), i-1) \;=\; P(FCED(e_2'), i-1)$

Proof. For any expression $F(e_1, e_2) \in \mathcal{L}_u^s$ and $i > 1$, we have

$$P(FCED(F(e_1, e_2)), i) = P(FCED(e_1), i-1) \times M_{i-1,i} \\ +P(FCED(e_2), i-1) \times N_{i-1,i}$$

Note that $P(FCED(e), i)$ does not contain any of the variables $\{M_{t,t+1}, N_{t,t+1} \mid i \leq t < k\}$. This means that the polynomial $P(FCED(F(e_1, e_2)), i)$ can be decomposed uniquely into the subpolynomials $P(FCED(e_1), i-1) \times M_{i-1,i}$ (which contains variable $M_{i-1,i}$), and $P(FCED(e_2), i-1) \times N_{i-1,i}$ (which contains variable $N_{i-1,i}$ but not $M_{i-1,i}$).

Lemma 3. *For any boolean assignment ρ, any FCED f, and any integer $i \in \{1, .., k\}$, we have*

$$P(||g, f||, i)[\rho] = P(f, i)[\rho], \text{ if } [\![g]\!]\rho = true \\ = 0, \qquad otherwise$$

Lemma 3 can be proved by induction on structure of FCED f.

Lemma 4. *For any boolean assignment ρ, any expression $e \in \mathcal{L}_u$, and any $1 \leq i \leq k$, we have*

$$P(FCED(e), i)[\rho] = P(FCED([\![e]\!]\rho), i)$$

Lemma 4 can be proved by induction on structure of expression e using Lemma 3.

We now state and prove the main theorem.

Theorem 2. *Let ρ be any boolean assignment, and let $e, e' \in \mathcal{L}_u$. Let $j = depth([\![e]\!]\rho)$ and let i be such that $j \leq i \leq k$. Then,*

$$[\![e]\!]\rho = [\![e']\!]\rho \text{ iff } P(FCED(e), i)[\rho] = P(FCED(e'), i)[\rho]$$

Proof. Let $e_\rho = [\![e]\!]\rho$ and $e'_\rho = [\![e']\!]\rho$. We prove that

$$e_\rho = e'_\rho \text{ iff } P(FCED(e_\rho), i) = P(FCED(e'_\rho), i)$$

and the result then follows from Lemma 4.

Suppose $e_\rho = e'_\rho$. We prove that $P(FCED(e_\rho), i) = P(FCED(e'_\rho, i)$. Note that $e_\rho \equiv e'_\rho$, i.e. e_ρ and e'_ρ must be syntactically identical. Hence, $P(FCED(e_\rho), i) = P(FCED(e'_\rho), i)$.

Now suppose that $P(FCED(e_\rho), i) = P(FCED(e'_\rho), i)$. We prove that $e_\rho = e'_\rho$. Note that e_ρ and e'_ρ have the same depth as $P(FCED(e_\rho, i))$ and $P(FCED(e'_\rho, i))$ have the same degree (since they are equal). The proof is by induction on structure of e_ρ. Note that $e_\rho \in \mathcal{L}_u^s$. Following cases arise.

- $e_\rho = x$
  This is the base case. Note that $P(FCED(e'_\rho, i)) = P(FCED(e_\rho, i)) = x$. Hence, $e'_\rho = x$.

- $e_\rho = F(e_1, e_2)$
  Let $e'_\rho = F(e'_1, e'_2)$. It follows from Property 1(a) and Property 1(b) that $P(FCED(e_1), i-1) = P(FCED(e'_1), i-1)$ and $P(FCED(e_2), i-1) = P(FCED(e'_2), i-1)$. Note that $depth(e_1) \leq j-1$ and $depth(e_2) \leq j-1$. Since $j-1 \leq i-1 \leq k$, we can apply the induction hypothesis to obtain $e_1 = e'_1$ and $e_2 = e'_2$. Thus, $e_\rho = e'_\rho$.

This completes the proof.

The proof of property T3 follows easily from Theorem 2.