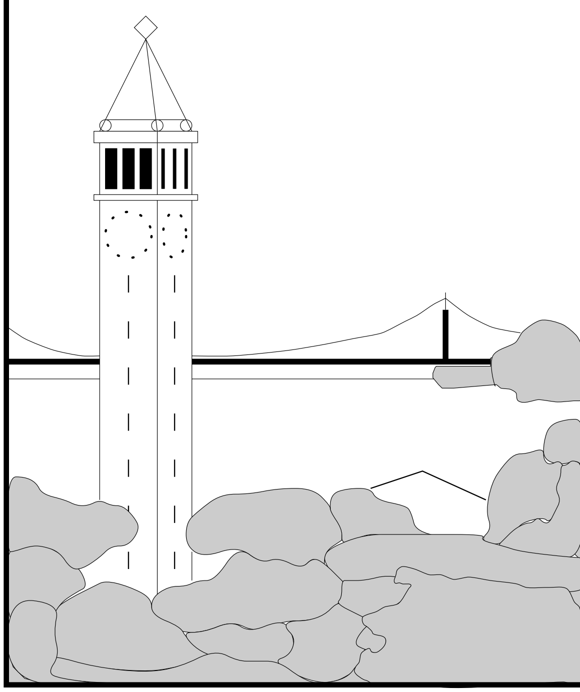
Global Value Numbering using Random Interpretation

(Full version)

Sumit Gulwani
gulwani@cs.berkeley.edu

George C. Necula
necula@cs.berkeley.edu



Report No. UCB/CSD-3-1296

November 2003

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

Global Value Numbering using Random Interpretation
(Full version)

Sumit Gulwani
gulwani@cs.berkeley.edu

George C. Necula
necula@cs.berkeley.edu

Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

Abstract

We present a polynomial time randomized algorithm for global value numbering. Our algorithm is complete when conditionals are treated as non-deterministic and all operators are treated as uninterpreted functions. We are not aware of any complete polynomial-time deterministic algorithm for the same problem. The algorithm does not require symbolic manipulations and hence is simpler to implement than the deterministic symbolic algorithms. The price for these benefits is that there is a probability that the algorithm can report a false equality. We prove that this probability can be made arbitrarily small by controlling various parameters of the algorithm.

Our algorithm is based on the idea of random interpretation, which relies on executing a program on a number of random inputs and discovering relationships from the computed values. The computations are done by giving random linear interpretations to the operators in the program. Both branches of a conditional are executed. At join points, the program states are combined using a random affine combination. We discuss ways in which this algorithm can be made more precise by using more accurate interpretations for the linear arithmetic operators and other language constructs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Theory, Verification

Keywords

Global Value Numbering, Herbrand Equivalences, Random Interpretation, Randomized Algorithm, Uninterpreted Functions

This research was supported in part by the National Science Foundation Career Grant No. CCR-9875171, and ITR Grants No. CCR-0085949 and No. 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.

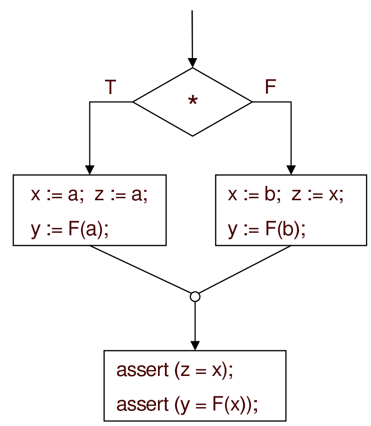


Figure 1. Example of non-trivial assertions

1 Introduction

Detecting equivalence of expressions in a program is a prerequisite for many important optimizations like constant and copy propagation [17], common sub-expression elimination, invariant code motion [3, 12], induction variable elimination, branch elimination, branch fusion, and loop jamming [9]. It is also important for discovering equivalent computations in different programs, for example, plagiarism detection and translation validation [11, 10], where a program is compared with the optimized version in order to check the correctness of the optimizer. Since the equivalence problem is undecidable, compilers typically implement algorithms that solve a restricted problem, where expressions are considered equivalent if and only if they are computed using the same operator applied on equivalent operands. This form of equivalence, where the operators are treated as uninterpreted functions, is called *Herbrand equivalence*. Such analyses, which include global value numbering [2], are widely used in optimizing compilers.

Existing algorithms for global value numbering are either too expensive or imprecise. The precise algorithms are based on an early algorithm by Kildall [8], where equivalences are discovered using an abstract interpretation [4] on the lattice of Herbrand equivalences. Kildall's algorithm discovers all Herbrand equivalences in a function body but has exponential cost [14]. On the other extreme, there are several polynomial time algorithms that are complete for basic blocks, but are imprecise in the presence of joins and loops in a program. An example of a program that causes difficulties is given in Figure 1.

The popular partition refinement algorithm proposed by Alpern, Wegman, and Zadeck (AWZ) [1] is particularly efficient, however at the price of being significantly less precise than the Kildall's algorithm. The novel idea in the AWZ algorithm is to represent the values of variables after a join using a fresh selection function ϕ_m, similar to the functions used in the static single assignment form [5], and to treat the ϕ_m function as another uninterpreted function. The values of z and x after the join in our example can both be written as $\phi_m(a,b)$. The AWZ algorithm then treats the ϕ functions as additional uninterpreted operators in the language and is able to detect that x and z are equivalent. The AWZ algorithm rewrites the second assertion as $\phi_m(F(a),F(b)) = F(\phi_m(a,b))$, which cannot be verified if the ϕ functions are uninterpreted.

In an attempt to remedy this problem, Rüthing, Knoop and Steffen have proposed a polynomial time algorithm that alternately applies the AWZ algorithm and some rewrite rules for normalization of terms involving ϕ functions, until the congruence classes reach a fixpoint [14]. Their algorithm discovers more equivalences than the AWZ algorithm (including the second assertion in our example). It is complete for acyclic control-flow graphs, but is incomplete in the presence of loops. Recently, Karthik Gargi has proposed a set of balanced algorithms that are efficient, but also incomplete [6].

In this paper, we describe a randomized algorithm that discovers as many Herbrand equivalences as the abstract interpretation algorithm of Kildall, while retaining polynomial time complexity. Our algorithm works by simulating the execution of a function on a small number of random values for the input variables. It executes both branches of a conditional, and combines the values of variables at join points using ϕ functions. The key idea is that each operator and each implicit ϕ function at a join point in the program is given a random interpretation. These interpretations are carefully chosen such that they obey all the semantic properties of ϕ functions (i.e. our algorithm does not regard ϕ functions as uninterpreted unlike the AWZ algorithm). This means that the values of variables computed in one pass through the program reflect all of the Herbrand equivalences that are common to all paths through the program. The algorithm is also simpler to implement than the deterministic symbolic algorithms, primarily because it resembles an interpreter that uses a simple mapping of variables to values as its main data structure. The price for the completeness and simplicity of the algorithm is that, in rare situations, the algorithm might report an apparent Herbrand equivalence that is actually false. We prove that the probability of this happening is very small.

The idea of giving random affine interpretations to ϕ functions has been used earlier in the context of a randomized algorithm for discovering linear equalities among variables in a program [7]. That algorithm, however, is limited to programs in which all computations consist of linear arithmetic. The biggest obstacle we had to overcome in trying to extend the linear arithmetic approach to arbitrary operators was to find a suitable class of random interpretations for the non-arithmetic operators. We show later in this paper that all straightforward interpretations (i.e., as some functions of the value of the operands) are either unsound or incomplete, when taken along with the affine interpretation of ϕ functions. Our solution is surprising because it requires several parallel simulations of the program. The result of an expression in a given simulation is not only based on its top-level operator and the values of its operands in that simulation, but also on the values of its operands in other simulations. We give a proof of probabilistic soundness and completeness of this scheme. We also give an analytical formula describing the number of parallel simulations required to achieve a

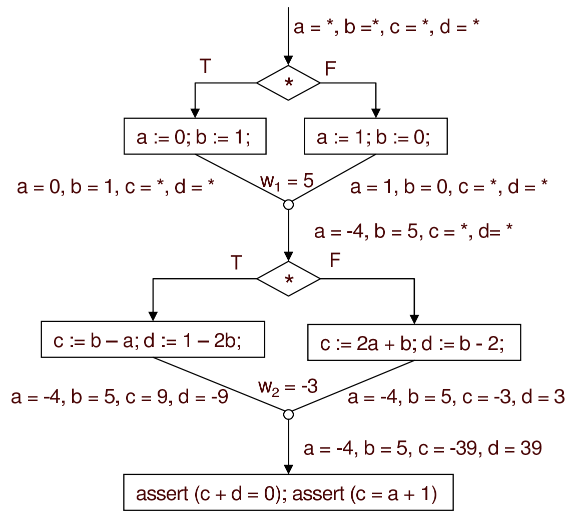


Figure 2. A code fragment with four paths. Of the two equations asserted at the end the first one holds on all paths but the second one holds only on three paths. The numbers shown next to each edge represent values of variables in the random interpretation scheme.

desired probability of error. Furthermore, we show that it is possible to combine, in the same algorithm, the natural interpretation of linear arithmetic operators with our random interpretation of non-arithmetic operators.

In Section 2 we review the random interpretation technique for discovering linear relationships. Then, in Section 3, we describe the proposed scheme for interpreting operators, and prove its soundness. In Section 4, we assemble the main ideas to construct the random interpreter for discovering Herbrand equivalences. In Section 5, we extend this scheme to discover more equivalences by using more accurate interpretations for the linear arithmetic operators and other language constructs.

2 Background

We illustrate the random interpretation scheme for discovering linear relationships among variables in a program [7], by means of an example. We also show a new proof of probabilistic soundness that gives insight into how this algorithm could be extended beyond linear arithmetic.

Consider the program shown in Figure 2 (ignoring for the moment the annotations shown on the side). Of the two assertions at the end of the program, the first is true on all four paths, and the second is true on three of them (it is false when the first conditional is false and the second is true). Regular testing would have to exercise that precise path to avoid inferring that the second equality holds. Instead, we use a non-standard interpretation model. At conditionals, we proceed on both true and false branches. At joins, we choose a random weight w and use it to combine the values v_1 and v_2 of a variable on the two sides of a join as follows:

$$\phi(v_1, v_2) = w \times v_1 + (1 - w) \times v_2$$

We call this operation an *affine join* of v_1 and v_2 with weight w, written as $v_1 \oplus_w v_2$. In essence, we are interpreting the ϕ functions as affine combinations with random weights.

In the example, all variables are dead on entry; so the random values with which we start the interpretation are irrelevant (we show them as $*$ in the figure). We use the random weights $w_1 = 5$ for the first join point and $w_2 = -3$ for the second join point. We perform the computations, maintaining at each step only a value for each variable. We can then verify easily that the resulting state at the end of the program satisfies the first assertion but does not satisfy the second. Thus, in one run of the program we have noticed that one of the exponentially many paths breaks the invariant. Note that choosing w to be either 0 or 1 at a join point corresponds to executing either the true branch or the false branch of its corresponding conditional; this is what naive random testing accomplishes. However, by choosing w (randomly) from a set that also contains non-Boolean values, we are able to capture the effect of both branches of a conditional in just one interpretation of the program.

The completeness argument of this interpretation scheme relies on the observation that by performing an affine join of two sets of values (all with the same weight), the resulting values satisfy all *linear* relationships that are satisfied by *both* initial sets of values. For the purpose of this paper, it is also important to note that (unfortunately) the affine join operation does not preserve non-linear relationships. For example, in the program in Figure 1 it is true that $a \times b = 0$, but this non-linear relationship is not implied by the program state after the first join point.

The probabilistic soundness argument given in [7] is complicated by an adjustment operation performed by the random interpreter. The purpose of this operation is to adjust a program state such that it reflects the additional equality fact implied by an equality conditional on its true branch. If we ignore this operation, we can give a simpler proof of soundness in terms of polynomials. A straight-line sequence of assignments, involving only linear arithmetic, computes the values of variables at the end as linear polynomials in terms of the variables live on input. The overall effect of the affine join operation is to compute the weighted sum of these polynomials corresponding to each path. These weights themselves are non-linear polynomials in terms of the random weights w_i. For example, the values of a, b, c and d at the end of the program shown in Figure 2 can be written as follows (there are no live input variables in this program):

$$\begin{array}{rcl}
a & = & w_1 \times 0 + (1-w_1) \times 1 \\
& = & 1 - w_1 \\
b & = & w_1 \times 1 + (1-w_1) \times 0 \\
& = & w_1 \\
c & = & w_2 \times (b-a) + (1-w_2) \times (2a+b) \\
& = & w_2 \times (w_1 - 1 + w_1) + (1-w_2) \times (2 - 2w_1 + w_1) \\
& = & 3w_1w_2 - w_1 - 3w_2 + 2 \\
d & = & w_2 \times (1-2b) + (1-w_2) \times (b-2) \\
& = & w_2 \times (1-2w_1) + (1-w_2) \times (w_1 - 2) \\
& = & -3w_1w_2 + w_1 + 3w_2 - 2
\end{array}$$

Correspondingly, the two assertions at the end of the program can be written, respectively, as $(3w_1w_2 - w_1 - 3w_2 + 2) + (-3w_1w_2 + w_1 + 3w_2 - 2) = 0$ and $3w_1w_2 - w_1 - 3w_2 + 2 = (1 - w_1) + 1$. Note that the first equality of polynomials is a tautology, while the second is not. We can prove that an assertion that is true on all paths (i.e., on all Boolean values for w_1 and w_2) will correspond to an equality between two equivalent polynomials. The opposite is true for assertions that are false on at least one path. Note that when fully expanded, these polynomials are exponential in size; however, this is not a problem since our interpreter can evaluate them in linear time.

The significance of reducing the problem to that of detecting polynomial equivalence lies in the following classic theorem due to Schwartz [15].

THEOREM 1 (RANDOMIZED POLYNOMIAL TESTING.). *Let $Q_1(x_1,..,x_n)$ and $Q_2(x_1,..,x_n)$ be two non-equivalent multivariate polynomials of degree at most d, in variables $x_1,..,x_n$ over a field $\mathcal{L}$. Fix any finite set $\tilde{L} \subseteq \mathcal{L}$, and let $a_1,..,a_n$ be chosen independently and uniformly at random from $\tilde{L}$. The probability that this choice is such that $Q_1(a_1,..,a_n) = Q_2(a_1,..,a_n)$ is at most $\frac{d}{|\tilde{L}|}$.*

Schwartz's theorem says that if a random evaluation of two polynomials returns the same result then it is very likely that the polynomials are equivalent. The theorem suggests that we can reduce the error probability in the random interpretation scheme by increasing the size of the set from which the random values are chosen. Additionally, the error probability decreases exponentially with the number of independent trials. Random testing can be thought of as an instance of this random interpretation scheme wherein the choice of weights w is restricted to the small set $\{0,1\}$ (this corresponds to executing either the true branch or the false branch of a conditional); but this gives a useless bound of $d/2$ for the error probability.

The lack of a known polynomial time deterministic algorithm for checking the equivalence of polynomials suggests that randomization has a chance to surpass deterministic algorithms in those program analysis problems that can be naturally reduced to checking equivalence of polynomials. Therefore it is not surprising that random interpretation works so well for checking equivalences in programs that involve only linear arithmetic computations. We show in the rest of this paper that even non-arithmetic operators can be encoded using polynomials. These schemes are not as obvious as for linear arithmetic. They also sacrifice precision since the precise meaning of the operator is lost. However, these schemes are very effective in discovering Herbrand equivalences.

3 Random Interpretation of Operators

We consider a language in which the expressions occurring in assignments and equality assertions belong to the following simple language of uninterpreted function terms (here x is one of the variables):

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

For simplicity, we consider only one binary uninterpreted function F. However, our results can be extended easily to languages with any finite number of uninterpreted functions of arbitrary arity. Comparing expressions in this language is trivial because only identical expressions are Herbrand equivalent. The complications arise in the presence of join points in a program as shown by the example in Figure 1.

The random interpreter compares expressions in this language by choosing an interpretation for F randomly from a suitable set of adequate interpretations, followed by choosing random values for the variables and evaluating the two expressions given these choices. We assume that the choice of the interpretation of F is made by choosing p parameters from some field $\mathcal{L}$. Thus, the interpretation of F, written $[\![F]\!]$ has the following type:

$$[\![F]\!] : \mathcal{L}^p \to \mathcal{L} \times \mathcal{L} \to \mathcal{L}$$

Given an expression e with n variables, the given interpretation of F induces an interpretation of the expression e:

$$[\![e]\!] : \mathcal{L}^p \to \mathcal{L}^n \to \mathcal{L}$$

We achieve the desired probabilistic soundness property of random interpretation by ensuring that, for random choices of $\pi \in \mathcal{L}^p$ and $\rho \in \mathcal{L}^n$, we have the following two properties (where $\overset{w.h.p.}{\Rightarrow}$, means "implies with high probability"):

$$[\![e_1]\!]\,\pi\,\rho \;=_{\mathcal{L}}\; [\![e_2]\!]\,\pi\,\rho \quad \overset{w.h.p.}{\Rightarrow} \quad [\![e_1]\!] \;=_{\mathcal{L}^p\to\mathcal{L}^n\to\mathcal{L}}\; [\![e_2]\!] \qquad (2)$$

$$[\![e_1]\!] \;=_{\mathcal{L}^p\to\mathcal{L}^n\to\mathcal{L}}\; [\![e_2]\!] \quad \Rightarrow \quad e_1 \;=\; e_2 \qquad (3)$$

We ensure property 2 by choosing the interpretation F to be a polynomial on $p+2$ variables. Assume now that we choose the following polynomial interpretation for F, with parameters r_1 and r_2:

$$[\![F]\!]\,(r_1,r_2)\,(x,y) \quad = \quad r_1x^2 + r_2y^2 \qquad (4)$$

This interpretation has the desired probabilistic soundness property, although the degree of the polynomial $[\![e]\!]$ is exponential in the depth of the expression e. According to Schwartz's theorem this drastically increases the probability of error, suggesting that perhaps we should consider only polynomials that are linear in the program variables (x and y).

There is, in fact, another important reason to choose linear polynomials. We choose the affine interpretation for ϕ functions because it is very effective in reasoning about linear expressions in a program [7]. We do not know of any other interpretation for ϕ functions that is effective in reasoning about any program properties. In order to ensure the desired completeness property of random interpretation, we require that $[\![F]\!]$ respects the affine interpretation given to the ϕ functions. This means that for all field values a, b, c, and d, and all $\pi \in \mathcal{L}^p$ we must have:

$$[\![\phi_m(F(a,b),F(c,d))]\!]\,\pi \quad \equiv \quad [\![F(\phi_m(a,c),\phi_m(b,d))]\!]\,\pi$$

or, equivalently:

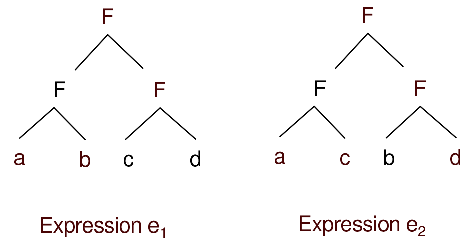
$$w[\![F]\!]\,\pi\,(a,b) + (1-w)[\![F]\!]\,\pi\,(c,d) \;\equiv\; [\![F]\!]\,\pi\,(wa+(1-w)c, wb+(1-w)d) \qquad (5)$$

It can be verified that the interpretation for F in equation 4 does not satisfy completeness property 5 (except for the cases when $w \in \{0,1\}$, which correspond exactly to the actual paths through the program). Moreover, it is possible to prove that if the ϕ functions are given the affine-join interpretation, and the completeness equation 5 is required to hold, then $[\![F]\!]\,\pi$ must be a linear polynomial in the program variables, for all values of $\pi \in \mathcal{L}^p$. The example in Section 2 that demonstrates that the affine join operation does not preserve non-linear relationships also illustrates this fact.

Unfortunately, if $[\![F]\!]\,\pi$ is a linear polynomial then the soundness equation 3 does not hold. Consider, for example, the linear interpretation

$$[\![F]\!]\,(r_1,r_2)\,(x,y) \quad = \quad r_1x + r_2y$$

In Figure 3 we show two distinct expressions that have the same interpretation, under this interpretation for F. Similar counterexamples arise for any linear interpretation, and in the presence of functions of arity at least two, but not if the language contains only unary functions, or constants.



e_1 = F(F(a,b),F(c,d))
$\quad = r_1[r_1(a)+r_2(b)] + r_2[r_1(c)+r_2(d)]$
$\quad = r_1^2(a) + r_1r_2(b+c) + r_2^2(d)$

e_2 = F(F(a,c),F(b,d))
$\quad = r_1[r_1(a)+r_2(c)] + r_2[r_1(b)+r_2(d)]$
$\quad = r_1^2(a) + r_1r_2(b+c) + r_2^2(d)$

Figure 3. An example of two distinct uninterpreted function terms e_1 and e_2 which are equivalent when we model the binary uninterpreted function F as a linear function of its arguments.

It appears that we have reached an impasse. If we fix the affine-join interpretation of ϕ, then only linear polynomials satisfy the completeness property. But linear polynomials are not sound interpretations of arbitrary operators. In the next section we describe a way out of this impasse.

3.1 Random k-Linear Interpretations

One way to characterize the failure of the soundness property when using linear interpretations for binary functions is that we are restricted to only three random coefficients, which are too few to encode a large number of leaves. Thus, it is possible for two distinct trees to have identical interpretations.

To increase the number of coefficients while maintaining linearity, we modify the interpreter to maintain k values for each variable and for each expression. This enables us to introduce more random parameters in the interpretation function. k is a parameter of the random interpreter, and we are going to derive lower bounds for k later in this section.

We need to refine the interpretations given in the previous section. Both the function F and any expression e now have a family of k interpretations, each with p parameters:

$$\begin{array}{rcl} [\![F]\!] & : & \{1,..,k\} \to \mathcal{L}^p \to \mathcal{L}\times\mathcal{L} \to \mathcal{L} \\ [\![e]\!] & : & \{1,..,k\} \to \mathcal{L}^p \to \mathcal{L}^n \to \mathcal{L} \end{array}$$

For the rest of the presentation we are going to work with a family of k linear polynomial interpretations (i.e. $[\![F]\!]\,i\,\pi$ is linear for all $1 \le i \le k$). This family uses $p = 4k-2$ parameters, named $r_1,..,r_k$, $r'_1,..,r'_k$, $s_1,..,s_{k-1}$ and $s'_1,..,s'_{k-1}$. In order to simplify the rest of the presentation, we introduce an alternate notation $P(e,i)$ for $([\![e]\!]\,i)$, for an expression e and index i between 1 and k. The definition of $P(e,i)$ is by induction on the structure of e, as follows:

$$\begin{array}{rcl} P(x,i) & = & x \\ P(F(e_1,e_2),1) & = & r_1 P(e_1,1) + r'_1 P(e_2,1) \\ P(F(e_1,e_2),i) & = & r_i P(e_1,i) + r'_i P(e_2,i) \\ & & +s_{i-1} P(e_1,i-1) \\ & & +s'_{i-1} P(e_2,i-1) \qquad \texttt{for } i > 1 \end{array}$$

Note that the degree of polynomial $P(e,i)$ is equal to the depth of expression e. Also note that for any i, $P(e,i)$ does not contain any of the variables $r_{i+1},..,r_k$ and $s_i,..,s_k$. This means that the polynomial $P(F(e_1,e_2),i)$ can be decomposed uniquely into the subpolynomials $r_i P(e_1,i) + r'_i P(e_2,i)$ (which contains variables r_i and r'_i), $s_{i-1} P(e_1,i-1)$ (which contains variable s_{i-1} but not r_i or r'_i), and $s'_{i-1} P(e_2,i-1)$ (which contains variable s'_{i-1} but not r_i, r'_i or s_{i-1}). This implies the following useful property.

PROPERTY 6. *For any integer* $i > 1$, $P(F(e_1,e_2),i) \equiv P(F(e'_1,e'_2),i)$ *iff*

(a) $r_i P(e_1,i) + r'_i P(e_2,i) \equiv r_i P(e'_1,i) + r'_i P(e'_2,i)$, *and*

(b) $P(e_1,i-1) \equiv P(e'_1,i-1)$, *and*

(c) $P(e_2,i-1) \equiv P(e'_2,i-1)$

We now prove the soundness lemma, which states that if the symbolic polynomials associated with two expressions are equivalent, then the two expressions are equal.

LEMMA 7 (K-LINEAR SOUNDNESS LEMMA). *Let e and e' be two tree expressions such that e has at most 2^j leaves, and $P(e,i) \equiv P(e',i)$ for some $i \geq j$. Then $e = e'$.*

PROOF. Note that e and e' have the same depth since $P(e,i)$ and $P(e',i)$ have the same degree (as they are equivalent). The proof is by induction on the structure of expressions e and e'. The base case is trivial since e and e' are both leaves and $P(e,i) = e$ and $P(e',i) = e'$. Clearly, $e = e'$.

For the inductive case, $e = F(e_1,e_2)$ and $e' = F(e'_1,e'_2)$. By assumption, $P(F(e_1,e_2),i) \equiv P(F(e'_1,e'_2),i)$ for some $i \geq j$. Since e has at most 2^j leaves, it must be that at least one of e_1 or e_2 has at most 2^{j-1} leaves. Consider the case when e_1 has at most 2^{j-1} leaves (the other case is symmetric). From Property 6(b) we have that $P(e_1,i-1) \equiv P(e'_1,i-1)$. Since $i-1 \geq j-1$, we can apply the induction hypothesis for e_1 and e'_1 to obtain that $e_1 = e'_1$. Consequently, $P(e_1,i) \equiv P(e'_1,i)$. This allows us to simplify the Property 6(a) to $P(e_2,i) \equiv P(e'_2,i)$. Since e_2 has at most 2^j leaves, we can apply the induction hypothesis for e_2 and e'_2 to conclude that $e_2 = e'_2$. This completes the proof. □

This result means that our family of interpretations is an injective mapping from trees to polynomials, and allows us to compare trees by random testing of their corresponding polynomials. Note that the higher the index of the polynomial, the larger the trees that it can discriminate. The number of parallel values that we need to compute for each node must be at least the logarithm of the number of leaves in the tree. Interestingly, this value does not depend on the depth of the tree. A consequence is that trees involving only unary constructors can be discriminated with $k = 1$, independent of the depth. The expressions that arise in programs can be represented as DAGs of size linear in the size of the program. In the worst case, the number of leaves in such a DAG, when expressed as a tree, is exponential in the largest depth of an expression computed by the program; thus k must be chosen at least as big as the largest depth of an expression computed by the program.

We have performed a number of experiments that suggest that an even tighter bound on k might be possible, but we are not able to prove any such result at the moment. We also have not been able to prove stronger properties by using more complex linear polynomial interpretations.

4 The Random Interpreter $\mathcal{R}$

We now put together the ideas mentioned in the previous sections to describe the random interpreter $\mathcal{R}$.

4.1 Notation

A *state* $\rho \in \mathcal{L}^n$ is an assignment of field values to the n variables of the program. We use the notation $\rho(x)$ to denote the value of variable x in state ρ. The notation $\rho[x \leftarrow q]$ denotes the state obtained from ρ by setting the value of variable x to q.

Our algorithm performs arithmetic over some field $\mathcal{L}$. For implementation reasons it is desirable that the field $\mathcal{L}$ should be finite so that arithmetic can be performed using finite representation for values. Hence, we choose $\mathcal{L} = \mathcal{Z}_q$, where q is some prime number and $\mathcal{Z}_q$ refers to the field containing the integers $\{0,..,q-1\}$. In this field, all the arithmetic operations are performed modulo prime q. The error probability of our algorithm is inversely proportional to the size of the field $\mathcal{L}$ (as stated in Theorem 15 in Section 4.4). Hence, by choosing a larger q, we can make the error probability of our algorithm smaller.

Our algorithm maintains k states at each point in the program, where k is as described in the previous section. This set of states is referred to as a *sample S*. We write S_i to refer to the i^{th} state of the sample S. The algorithm computes k values for each expression e in the program. The i^{th} value of an expression e at some program point α can be written as $P(e,i)\ \pi\ S_i$, where P is the polynomial interpretation given in the previous section, π refers to the values of the parameters r_i, r'_i, s_{i-1} and s'_{i-1} chosen independently and uniformly at random from the finite field $\mathcal{L}$, and S is the sample at the program point α. However, this value is computed directly (without first computing the polynomial $P(e,i)$) by simply evaluating the expression on the given state and the values chosen for the parameters r_i, r'_i, s_{i-1} and s'_{i-1}. Essentially, this value is computed by the function $V(e,i,S)$, whose definition is given below.

$$\begin{array}{rcl} V(x,i,S) & = & S_i[x] \\ V(F(e_1,e_2),1,S) & = & r_1 V(e_1,1,S) + r'_1 V(e_2,1,S) \\ V(F(e_1,e_2),i,S) & = & r_i V(e_1,i,S) + r'_i V(e_2,i,S) \\ & & +s_{i-1} V(e_1,i-1,S) \\ & & +s'_{i-1} V(e_2,i-1,S)\ for\ i > 1 \end{array}$$

We say that a sample S satisfies a Herbrand equivalence $e_1 = e_2$ when $V(e_1,k,S) = V(e_2,k,S)$. We write $S \models e_1 = e_2$ when this is the case. Note that we use only the k^{th} value when deciding the equivalence. This is motivated by Lemma 7, which says that the k^{th} polynomial has the most discriminating power among the polynomials that we evaluate.

Finally, we extend the affine join operation from individual values to states, in which case we perform the join with the same weight for each variable. We further extend the affine join operation to samples, in which case we perform the affine join operation on each pair of corresponding states with the same weight.

4.2 The Random Interpreter Algorithm

The random interpreter $\mathcal{R}$ executes a procedure like an abstract interpreter or a data-flow analyzer. It goes around each loop until a fixed point is reached. The criterion for fixed point is defined in Section 4.4. The random interpreter maintains a sample of k states at each program point. These samples encode Herbrand equivalences, or relationships among uninterpreted function terms of a program. A sample at a program point is obtained from the sample(s) at the immediately preceding program point(s). The initial sample consists of k copies of a randomly chosen state ρ, i.e. the values of all variables of the program in state ρ are chosen independently and uniformly at random from the field $\mathcal{L}$. We now describe the action of the random interpreter on the three basic nodes of a flow-chart, which are shown in Figure 4.

- Assignment Node: See Figure 4 (a).
  $S_i = S_i'[x \leftarrow V(e,i,S)]$
- Conditional Node: See Figure 4 (b).
  $S^1 = S'$ and $S^2 = S'$
- Join Node: See Figure 4 (c).
  $S = S^1 \oplus_w S^2$, where w is a fresh random value chosen independently and uniformly at random from $\mathcal{L}$.

After fixed point has been reached, the results of the random interpreter can be used to verify or discover equivalences among expressions at any point in the program as follows. Two expressions e_1 and e_2 always have the same value at some point P in a program if $S \models e_1 = e_2$, where S is the sample at point P, or equivalently if their k^{th} value is the same in the given sample.

4.3 Completeness and Soundness Theorems

For the purpose of the analysis of the algorithm, we introduce two new interpreters: a symbolic random interpreter $\tilde{\mathcal{R}}$, which is a symbolic version of the random interpreter $\mathcal{R}$, and an abstract interpreter $\mathcal{A}$, which is sound and complete. We prove that $\tilde{\mathcal{R}}$ is as complete and as sound as $\mathcal{A}$. We then show that this implies that $\mathcal{R}$ is probabilistically sound, and is complete when all operators are uninterpreted and the conditionals are non-deterministic.

4.3.1 The Symbolic Random Interpreter $\tilde{\mathcal{R}}$

The symbolic random interpreter $\tilde{\mathcal{R}}$ maintains a symbolic state and executes a program like the random interpreter $\mathcal{R}$ but symbolically. Instead of using random values for the initial values for variables, or the parameters w, r_i and s_i, it uses variable names and maintains symbolic expressions. We use the letter $\tilde{S}$ to range over the symbolic samples maintained by the symbolic random interpreter. We write $\tilde{V}(e,i,\tilde{S})$ to denote the i^{th} symbolic value of expression e in symbolic sample $\tilde{S}$.

The following property states the relationship between the samples computed by $\mathcal{R}$ and the symbolic samples computed by $\tilde{\mathcal{R}}$.

PROPERTY 8. *Let $\tilde{S}$ be a symbolic sample computed by $\tilde{\mathcal{R}}$ at some point in the program and let S be the corresponding sample computed by $\mathcal{R}$ at the same point. The sample S can be obtained from the symbolic sample $\tilde{S}$ by substituting the input variables, the weight and parameter variables w, r_i and s_i with the values that $\mathcal{R}$ has used for them.*

4.3.2 The Abstract Interpreter $\mathcal{A}$

The abstract interpreter $\mathcal{A}$ computes the Herbrand equivalences in a program. In the following definition we use the letter U to range over sets of Herbrand equivalences. We write $U \Rightarrow e_1 = e_2$ to say that the conjunction of the Herbrand equivalences in U imply $e_1 = e_2$. We write $U_1 \cap U_2$ for the set of Herbrand equivalences that are implied by both U_1 and U_2. Finally, we write $U[e/x]$ for the relationships that are obtained from those in U by substituting e for x. With these definitions we can define the action of $\mathcal{A}$ over the nodes of a flow-chart as follows:

- Assignment Node: See Figure 4 (a).
  $U = \{x = e[x'/x]\} \cup U'[x'/x]$, where x' is a fresh variable
- Conditional Node: See Figure 4 (b).
  $U^1 = U'$ and $U^2 = U'$
- Join Node: See Figure 4 (c).
  $U = U^1 \cap U^2$

The abstract interpreter starts with the empty set of Herbrand equivalences. Implementations of abstract interpretations such as $\mathcal{A}$ have been described in the literature [8]. The major concern there is the concrete representation of the set U and the implementation of the operation $U_1 \cap U_2$. In Kildall's original presentation the set U has an exponential-size representation, although this is not necessary [14]. Here we use $\mathcal{A}$ only to state and prove the soundness and completeness results of the random interpreter $\mathcal{R}$. The abstract interpreter $\mathcal{A}$ is both sound and complete when all operators are uninterpreted and conditionals are non-deterministic [16, 14].

We now state the relationship between the sets of symbolic samples $\tilde{S}$ computed by $\tilde{\mathcal{R}}$ and the sets of Herbrand equivalences U computed by $\mathcal{A}$ in the form of completeness and soundness theorems.

THEOREM 9 (COMPLETENESS THEOREM). *Let U be a set of Herbrand equivalences computed by $\mathcal{A}$ at some point in the program and let $\tilde{S}$ be the corresponding symbolic sample. Let e_1 and e_2 be any two expressions such that $U \Rightarrow e_1 = e_2$. Then, $\tilde{S} \models e_1 = e_2$.*

The completeness theorem implies that the random interpreter $\mathcal{R}$ discovers all the Herbrand equivalences that the abstract interpreter $\mathcal{A}$ discovers. The proof of Theorem 9 is based on Lemma 10 which is stated and proved below. Lemma 10 states that the affine join of two states satisfies all the Herbrand equivalences that are satisfied by both the states. The full proof of Theorem 9 is given in Appendix A.1.

LEMMA 10 (UNION COMPLETENESS LEMMA). *Let $\tilde{S}$ and $\tilde{S}'$ be two symbolic samples that satisfy the Herbrand equivalence $e_1 = e_2$. Then, for any choice of weight w, the union $\tilde{S}^u = \tilde{S} \oplus_w \tilde{S}'$ also satisfies the same Herbrand equivalence.*

PROOF. Note that for any expression e, and any symbolic sample $\tilde{T}$, $\tilde{V}(e,k,\tilde{T})$ is a linear function of the program variables in expression e. Hence, for any affine combination of two symbolic states $\tilde{S} \oplus_w \tilde{S}'$, one can easily verify that $\tilde{V}(e,k,\tilde{S} \oplus_w \tilde{S}') = w \times \tilde{V}(e,k,\tilde{S}) + (1-w) \times \tilde{V}(e,k,\tilde{S}')$. Thus, if $\tilde{V}(e_1,k,\tilde{S}) = \tilde{V}(e_2,k,\tilde{S})$ and $\tilde{V}(e_1,k,\tilde{S}') = \tilde{V}(e_2,k,\tilde{S}')$, then $\tilde{V}(e_1,k,\tilde{S} \oplus_w \tilde{S}') = \tilde{V}(e_2,k,\tilde{S} \oplus_w \tilde{S}')$. From here the completeness statement follows immediately. $\square$

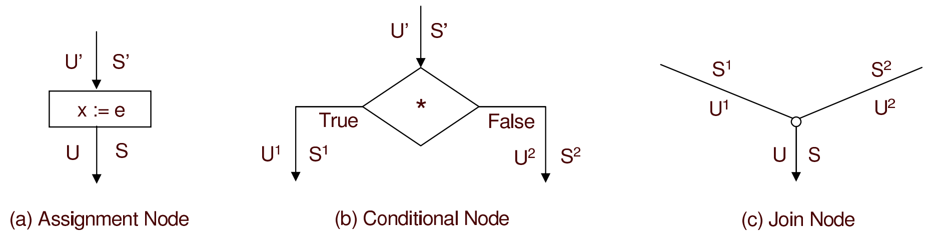


Figure 4. Flow-chart nodes

It is not surprising that the completeness lemma holds, since we have chosen the linear interpretations of operators specifically to satisfy this constraint. Next we state the soundness theorem.

THEOREM 11 (SOUNDNESS THEOREM). *Let U be a set of Herbrand equivalences computed by $\mathcal{A}$ at some point in the program and let $\tilde{S}$ be the corresponding symbolic sample of k symbolic states. Let e_1 and e_2 be two expressions such that $\tilde{S} \models e_1 = e_2$, and $k \geq min(degree(\tilde{V}(e_1,k,\tilde{S})), degree(\tilde{V}(e_2,k,\tilde{S})))$. Then, $U \Rightarrow e_1 = e_2$.*

According to Theorem 11, if the symbolic polynomials associated with two expressions under our random interpretation scheme are equivalent, then those two expressions are also found equivalent by the abstract interpreter. The proof of Theorem 11 is based on Lemma 7. Notice, however, that in Theorem 11 the lower bound on k is stated based on the degree of $\tilde{V}(e,k,\tilde{S})$, which is equal to the depth of expression e, while in Lemma 7, it is based on the logarithm on the number of leaves. The reason for this weakening of the soundness statement is two-fold: it would have been more complicated to carry out the proof with leaf counts, and in the worst case these measures are equal. The full proof of Theorem 11 is given in Appendix A.2. We use this soundness theorem in the next section to prove Theorem 15, which establishes an upper bound on the probability that $\mathcal{R}$ is unsound.

4.4 Fixed Point Computation

For a program with loops, the random interpreter $\mathcal{R}$ goes around each loop until a fixed point is reached, like an abstract interpreter or a dataflow analysis algorithm. A fixed point is reached when the Herbrand equivalences inferred from the numerical results of $\mathcal{R}$ are stable. The main concerns are then whether the fixed-point is ever reached, and how many iterations are required. The answers to these questions are implied by the fact that the lattice of sets of Herbrand equivalences that are true at any point in a program has finite depth as stated in Theorem 12 below.

THEOREM 12. *The lattice of sets of Herbrand equivalences (involving the program variables) that are true at any point in a program (under the set union operation as described in Section 4.3.2) has depth at most n where n is the number of program variables.*

The proof of Theorem 12 follows easily from Lemma 13 and Property 14. Before stating them, we first introduce some useful notation. Let T be the set of program variables. Let $\preceq$ be any total order on the program variables. We use the notation $y \prec z$ to denote that $y \preceq z$, and that y and z are distinct variables. For notational convenience, we also say that $F(e_1,e_2) \prec y$ for any function term $F(e_1,e_2)$ and variable y. For any expression e, we use the notation $Vars(e)$ to denote the set of variables that occur in expression e.

LEMMA 13. *The Herbrand equivalences at any point in the program can be represented by a pair $H = (I,E)$, where $I \subseteq T$ is a set of independent variables and E is a set of equivalences $x = e$, one for each variable $x \in T - I$, such that $Vars(e) \subseteq I$ and $e \prec x$.*

The proof of this lemma is by induction on structure of the program and is given in Appendix B.

PROPERTY 14. *Suppose (I_2,E_2) is above (I_1,E_1) in the lattice (which is to say that E_2 is a weaker set of equivalences than E_1). Then $I_2 \supset I_1$.*

PROOF. We first make two useful observations. Let the Herbrand equivalences at any point in the program be represented by the pair (I,E). Then,
(a) E does not imply any equivalence of the form $x = e$ for any variable $x \in I$ and any expression e such that $e \prec x$.
(b) E does not imply any equivalence of the form $e_1 = e_2$ such that $Vars(e_1) \subseteq I$, $Vars(e_2) \subseteq I$ and e_1 is not identical to e_2.

We first show that $I_2 \supseteq I_1$. Suppose for the purpose of contradiction that $I_2 \not\supseteq I_1$. Then, E_2 contains an equality of the form $x = e$ for some variable $x \in I_1$ and expression e such that $e \prec x$. Since E_1 is a stronger set of equivalences, E_1 implies $x = e$. But this is not possible because of observation (a) above.

We now show that $I_2 \supset I_1$. Suppose for the purpose of contradiction that $I_2 = I_1$. Since E_1 is a stronger set of equivalences than E_2, E_1 contains an equivalence $x = e_1$, where $x \in T - I_1$ and $Vars(e_1) \subseteq I_1$, such that $E_2 \not\Rightarrow x = e_1$. Note that $x \in T - I_2$ since $I_2 = I_1$. Hence, E_2 contains an equivalence $x = e_2$, where $Vars(e_2) \subseteq I_2$. Note that $e_1 \not\equiv e_2$ since $E_2 \not\Rightarrow x = e_1$ and $E_2 \Rightarrow x = e_2$, Since E_1 is a stronger set of equivalences than E_2, $E_1 \Rightarrow x = e_2$ and hence $E_1 \Rightarrow e_1 = e_2$. But this is not possible because of observation (b) above. □

Thus, the abstract interpreter $\mathcal{A}$ is guaranteed to reach a fixed point within a number of iterations that is linear in the number of variables of the program. Given the close relationship between $\mathcal{A}$ and $\tilde{\mathcal{R}}$ as established by Theorem 9 and Theorem 11, $\tilde{\mathcal{R}}$ also reaches a fixed point in the same number of loop iterations. Furthermore, given the relationship between $\tilde{\mathcal{R}}$ and $\mathcal{R}$ as mentioned in Property 8, $\mathcal{R}$ also reaches a fixed point with high probability in the same number of loop iterations.

The above observations suggest that $\mathcal{R}$ must go around each loop for n steps (this would guarantee that fixed-point has been reached), where n is the number of variables that are defined inside the loop. Another alternative is to detect when fixed-point has been reached by comparing the set of Herbrand equivalences implied by $\mathcal{R}$ in two successive executions of a loop (this can be done by building a symbolic value flow graph of the program [13]). If these sets are identical, then the fixed-point for that loop has been reached.

An upper bound on the number of iterations required for reaching fixed-point enables us to state an upper bound on the error probability in the analysis performed by the random interpreter $\mathcal{R}$.

THEOREM 15 (PROBABILISTIC SOUNDNESS THEOREM). *Let e_1 and e_2 be two non-equivalent expressions of depth at most t at some program point. Let S be the random sample at that program point after fixed-point. If $k \geq 2n^2 + t$, then $\Pr[S \models e_1 = e_2] \leq \frac{2n^2+t}{|\mathcal{L}|}$, where n is an upper bound on the number of variables, function applications, and join points in the program, and $|\mathcal{L}|$ denotes the size of the field $\mathcal{L}$ from which the random values are chosen.*

PROOF. Let $\tilde{S}$ be the corresponding symbolic sample, and U be the corresponding set of Herbrand equivalences at that point. Since the abstract interpreter $\mathcal{A}$ is sound, $U \not\Rightarrow e_1 = e_2$. There are at most n function applications and at most n join points in the program. Each function application and each join operation increases the degree of the polynomial corresponding to the resulting expression by 1. Hence, one loop iteration contributes $2n$ to the degree of the polynomial corresponding to an expression. The fixed point computation requires at most n iterations. Hence, the degrees of the polynomials $\tilde{V}(e_1, k, \tilde{S})$ and $\tilde{V}(e_2, k, \tilde{S})$ are bounded above by $2n^2 + t$. It thus follows from Theorem 11 that $\tilde{S} \not\models e_1 = e_2$. The desired result now follows from Property 8 and Theorem 1. $\square$

Theorem 15 implies that by choosing $\mathcal{L}$ big enough, the error probability can be made as small as we like. In particular, if $n < 100$, and if we choose $\mathcal{L}$ such that $|\mathcal{L}| \approx 2^{32}$ (which means that the random interpreter can perform arithmetic using 32-bit numbers), then the error probability is bounded above by 10^{-5}. By repeating the algorithm b times, the error probability can be further reduced to 10^{-5b}.

4.5 Computational Complexity

The cost of each assignment operation performed by the random interpreter is $O(k)$, assuming that each assignment operation involves a constant number of function applications. Let n be the number of assignments in a program. The fixed-point for any loop is reached in at most n steps. Therefore, the total number of assignment operations performed by the random interpreter is at most $O(n^2)$. Thus, the total cost of all assignment operations is $O(n^2 k)$. The cost of a single join operation is $O(mk)$, where m is the number of ϕ assignments at the join point. The total cost of all join operations can be amortized to $O(n^2 \times k)$ (since each ϕ assignment can be associated with an ordinary assignment). Hence, the total time taken by the random interpreter is $O(n^2 \times k)$. Choosing $k = O(n^2)$, in order to satisfy the requirement for probabilistic soundness, yields an overall complexity of $O(n^4)$.

Our analysis for probabilistic soundness requires choosing $k = O(n^2)$. However, we feel that our analysis is very conservative. The experiments that we have performed also suggest that tighter bounds on k might be possible, but we are not able to prove any such result at the moment. Note that Lemma 7 requires working with only $\log n$ polynomials, where n is the size of the tree expressions. If we can prove a similar lemma for DAGs, then we can prove that choosing $k = O(\log n)$ is sufficient for probabilistic soundness, which will yield an overall complexity of $O(n^2 \log n)$ for our algorithm.

5 Beyond Herbrand Equivalences

Until now, we have discussed how to discover equivalences in a program in which all the operators are treated as uninterpreted. More equivalences can be discovered if some of these operators are interpreted.

5.1 Linear Arithmetic

The random interpretation scheme described in our earlier paper [7] discovers all linear relationships among variables in a program in which all assignments compute only linear arithmetic expressions. The random interpretation scheme described in this paper discovers all Herbrand equivalences in a program in which all operators are treated as uninterpreted. It is interesting to consider whether by combining both these schemes, we can discover all the equivalences in a program that has expressions consisting of both linear arithmetic as well as uninterpreted operators as described below (here q denotes a rational number):

$$e ::= x \mid F(e_1, e_2) \mid q \mid q \times e \mid e_1 \pm e_2$$

One way to combine both these schemes is to extend the description of the random interpreter $\mathcal{R}$ to use the natural interpretation for the linear arithmetic operators as follows:

$$\begin{array}{rcl} V(q, i, S) & = & q \\ V(q \times e, i, S) & = & q \times V(e, i, S) \\ V(e_1 \pm e_2, i, S) & = & V(e_1, i, S) \pm V(e_2, i, S) \end{array}$$

Such a naive combination of the two schemes is unsound. For example, consider the two non-equivalent expressions $e_1 = F(a,b) + F(c,d)$ and $e_2 = F(a,d) + F(c,b)$. It is easy to see that for any sample S and any i, $V(e_1, i, S) = V(e_2, i, S)$.

One way to fix this problem is to hash the value of an uninterpreted function term before being used in an arithmetic expression. This loses some information about the uninterpreted term, but prevents the unintended interaction between the chosen linear interpretation of the operator and the plus operator. For this purpose, we maintain an extra bit of information with every variable in a sample, namely whether the top-level operator used in computing the value of a variable was an uninterpreted operator or not. The random interpreter $\mathcal{R}$ now maintains a tuple (Q, S) at every point in the program, where S refers to a sample as before, and Q is a mapping that maps every variable to some Boolean value. The random interpreter updates Q as follows:

- Assignment Node: See Figure 4 (a).
  $Q = Q'[x \leftarrow \texttt{True}]$ if e is of the form $F(e_1, e_2)$
  $\quad = Q'[x \leftarrow \texttt{False}]$, otherwise
  where Q' refers to the mapping before the assignment node, and Q refers to the mapping after the assignment node.
- Conditional Node: See Figure 4 (b).
  $Q^1 = Q'$ and $Q^2 = Q'$
  where Q' refers to the mapping before the conditional node, and Q^1 and Q^2 refer to the mappings after the conditional node.
- Join Node: See Figure 4 (c).
  $Q(x) = Q^1(x) \vee Q^2(x)$
  where Q^1 and Q^2 refer to the mappings before the join node, and Q refers to the mapping after the join node.

The function V now assigns values to linear arithmetic expressions in the following manner.

$$\begin{aligned} V(e_1 \pm e_2, i, S) &= \mathit{ToArith}(e_1, i, S) \pm \mathit{ToArith}(e_2, i, S) \\ V(q \times e, i, S) &= q \times \mathit{ToArith}(e, i, S) \end{aligned}$$

where *ToArith* is a function that hashes uninterpreted function terms as follows:

$$\begin{aligned} \mathit{ToArith}(e_1, i, S) &= \texttt{if } Q(e_1) \texttt{ then } V(e_1, i, S) \\ & \quad\; \texttt{else } \mathit{Hash}(V(e_1, i, S)) \end{aligned}$$

Such a random interpretation scheme is probabilistically sound but not complete. For example, consider the two equivalent expressions $e_1 = (F(a,b)+c)-c$ and $e_2 = F(a,b)$. It is easy to see that for any sample S and any i, $V(e_1,i,S) \neq V(e_2,i,S)$ (with high probability over the random choices made by the Hash function) since $V(e_1,i,S) = \mathit{Hash}(V(F(a,b),i,S))$ and $V(e_2,i,S) = V(F(a,b),i,S)$. We can increase the precision with a simple modification. We can convert an arithmetic value back to an uninterpreted function term value if the arithmetic value is equal to the hash of an uninterpreted function term value. This modification can discover all equivalences inside basic blocks, but still remains incomplete for discovering equivalences across basic blocks. It is an interesting question to figure out if there exists an efficient algorithm that discovers all the equivalences in the presence of linear arithmetic, uninterpreted operators and non-deterministic conditionals.

5.2 Bitwise Operations

If we attempt to use the natural interpretation for non-arithmetic operators as well, we loose probabilistic soundness. This is because non-arithmetic expressions cannot be expressed as polynomials. For example, consider the following program fragment in which x, y, and z are input variables that take integral values, and & denotes the "bitwise and" operator.

```
t := x & y & z & 1;
assert (t = 0)
```

The assertion is not always true, yet if the basic block is executed with values for x, y, and z chosen randomly from $\mathcal{L}$, then the probability that the assert statement at the end of the program is falsified is $\frac{1}{8}$ (the probability that all x, y and z are all chosen to be odd integers), meaning that most likely the random interpreter will erroneously validate the assert. Note that this problem is related to the NP-complete problem of detecting equivalences of Boolean expressions. One should not expect that equivalence of Boolean expressions can be decided simply by random testing since the problem of deciding equivalence of Boolean expressions is NP-complete, and it is not known whether NP-complete problems can be decided in randomized polynomial time (RP).

One way to conservatively handle the bitwise operators is to model them as uninterpreted functions. However, this may be too conservative. Another way is to interpret the bitwise logical operators using the multiplication and addition operators as described below.

$$\begin{aligned} V(e_1 \& e_2, i, S) &= V(e_1 * e_2, i, S) \\ V(e_1 | e_2, i, S) &= V(e_1 + e_2, i, S) \end{aligned}$$

Here $|$ denotes the "bitwise or" operator. This interpretation captures the commutativity and associativity properties of the "bitwise and" and "bitwise or" operators inside basic blocks. It also captures the distributivity of "bitwise and" operator over the "bitwise or" operator. However, this is still far from capturing all the Boolean axioms. For example, it does not capture the distributivity of "bitwise or" operator over the "bitwise and" operator. It also does not capture the axiom that $e \,\&e \equiv e$. Note that this interpretation is sound because & and $|$ satisfy all the properties that are satisfied by $*$ and $+$ respectively.

The random interpretation scheme for handling arithmetic is as expected:

$$\begin{aligned} V(e_1 * e_2, i, S) &= V(e_1, i, S) * V(e_2, i, S) \\ V(e_1 + e_2, i, S) &= V(e_1, i, S) + V(e_2, i, S) \end{aligned}$$

This is a sound scheme since the random interpretation scheme is sound for testing equivalence of polynomials.

The above interpretation will be unsound if expressions involve both the bitwise operations and the arithmetic operations. Hence, an expression consisting of "bitwise and" and "bitwise or" operator must be hashed before being used in an arithmetic expression, just as uninterpreted function terms are hashed before being used in an arithmetic expression as described in Section 5.1.

5.3 Memory Reads and Writes

As another interesting example, consider the following program fragment in which x and y are input variables that take integer values, and *Mem* refers to some array. Note that the assert statement at the end of the program is not always true (since the input variables x and y may have the same values).

```
Mem[x] := 0;
Mem[y] := 1;
t := Mem[x];
assert (t = 0)
```

If the basic block is executed with values for x and y chosen randomly from $\mathcal{L}$, then the probability that the assert statement at the end of the program is falsified (and the random interpreter detects the possible failure of the assert) is $\frac{1}{|\mathcal{L}|}$, which is very small. This is the probability that x and y are chosen equal. Again, the problem of detecting equivalences of expressions inside a basic block involving memory reads and writes is NP-hard. Hence, one should not expect to decide equivalences of such expressions simply by random testing.

Memory reads and memory writes can be modeled conservatively using two special uninterpreted functions as follows. The memory state is represented by a special variable μ. A memory read is modeled by the binary uninterpreted function `Select` that takes as arguments the state of the memory (represented by the special variable μ) and the address in the memory as follows.

$$V(\mathit{Mem}[e], i, S) = V(\texttt{Select}(\mu, e), i, S)$$

A memory write $\mathit{Mem}[e_1] = e_2$ is modeled by updating the value of the special memory variable μ in the sample before the assignment as follows.

$$S_i = S_i'[\mu \leftarrow V(\texttt{Update}(\mu, e_1, e_2), i, S)]$$

Here `Update` is an uninterpreted function of 3 arguments. Such a select-update formalism is commonly used to model memory and has also been used in [10]. Treating `Select` and `Update` operators as uninterpreted helps to reason at least about the fact that two reads from the same memory location with no intervening memory writes yield same values.

5.4 Integer Division Operator

As another example, consider the following program fragment in which x is an input variable that takes integral values, and / denotes the integer division operator.

```
t := 2 ÷ x;
assert (t = 0)
```

The assert statement above is not always true. If the basic block is executed with values for x chosen randomly from $\mathcal{L}$, then the probability that the assert statement at the end of the program is falsified is at most $\frac{2}{|\mathcal{L}|}$, which is very small. This is the probability that x is chosen to be either 1 or 2. Note that this problem can be easily reduced to the problem of checking whether a polynomial has integral roots or not, which is an undecidable problem. Hence, one should not expect that equivalences of expressions involving the integer division operator can be decided by random testing.

One way to conservatively handle the integer division operator is to model it as an uninterpreted function. However, this may be too conservative. Another way to handle the integer division operator is to model it as a real division operator as follows.

$$V(e_1 \div e_2, i, S) \quad = \quad V(\frac{e_1}{e_2}, i, S)$$

Here $\frac{a_1}{a_2}$ denotes the real division of a_1 by a_2, while $a_1 \div a_2$ denotes the integer division of a_1 by a_2. This modeling, though incomplete, may help in detecting equivalences that are not discovered by the earlier one. This is a sound modeling because if $\frac{e_1}{e_2} = \frac{e_3}{e_4}$, then $e_1 \div e_2 = e_3 \div e_4$.

The random interpretation scheme for handling real division is as follows.

$$V(\frac{e_1}{e_2}, i, S) \quad = \quad \frac{V(e_1, i, S)}{V(e_2, i, S)}$$

Note that this is a sound scheme because the random interpretation scheme is sound for testing equivalence of polynomials, and if $e_1 \times e_4 = e_3 \times e_2$, then it must be the case that $\frac{e_1}{e_2} = \frac{e_3}{e_4}$.

6 Comparison with Related Work

The algorithms for global value numbering in literature can be broadly classified into being optimistic or pessimistic.

The optimistic algorithms start with the optimistic assumption that the expressions not known to be unequal are equal. Hence they may maintain some incorrect facts about a program at an intermediate analysis stage [8]. These assumptions get refined in successive loop iterations, and the process is repeated until the assumptions become consistent. The precise algorithms that discover all Herbrand equivalences in a function body fall into this optimistic category. They are based on an early algorithm by Kildall [8], which discovers equivalences using an abstract interpretation on the lattice of Herbrand equivalences. The running time of these algorithms is exponential. Our algorithm also falls under the optimistic category. It is based on random interpretation on the lattice of Herbrand equivalences. It is complete and discovers all the equivalences that are discovered by the Kildall's algorithm. However, our algorithm runs in polynomial time $O(n^4)$.

The pessimistic algorithms start with the pessimistic assumption that the expressions not known to be equal are unequal. These algorithms maintain only true facts about a program at every intermediate stage of the analysis and do not require fixed-point computation. These algorithms are based on the popular partitioning algorithm by Alpern, Wegman, and Zadeck [1], which runs in time $O(e \log n)$ where n and e are the number of nodes and edges in a procedure's static single assignment graph. The running time of these algorithms is better than our algorithm because the pessimistic assumption does not need to be refined or reapplied, while the optimistic assumption must be repeatedly refined and reapplied until it becomes consistent. But, these algorithms cannot discover all Herbrand equivalences in a procedure and hence are less precise than our algorithm. The preciseness of the results of our algorithm may outweigh its execution time.

Recently, there have been proposals for a hybrid approach which tries to combine the best of both the above approaches. Karthik Gargi has proposed *balanced algorithms* which start with optimistic assumptions for the reachability of blocks and edges and the pessimistic assumption for the congruence of values [6]. He has demonstrated experimentally that balanced algorithms terminate faster than the optimistic algorithms and produce more precise information than the pessimistic algorithms. Oliver Rüthing, Jens Knoop and Bernhard Steffen have extended the partition refinement algorithm proposed by Alpern, Wegman and Zadeck [1] with the concept of *integrated normalization* [14], wherein the partitioned value graphs are modified according to a set of graph rewrite rules and the process (of partitioning the value graph and modifying it) is repeated until fixed-point is reached. The graph rewrite rules are able to discover some more equivalences which further trigger detection of equivalences by the partitioning algorithm. These hybrid algorithms discover more equivalences than the pessimistic algorithms, but they also cannot discover all Herbrand equivalences and are less precise than our algorithm. The running time of the algorithm by Rüthing, Knoop and Steffen is $O(n^4 \log n)$ and is comparable to the running time of our algorithm, which is $O(n^4)$.

The random interpretation scheme described in this paper shares the idea of using an affine-join interpretation for ϕ functions with the algorithm described in [7]. We extend that idea with a way to interpret uninterpreted operators in the language in a manner that is complete and probabilistically sound. Furthermore, we show how to improve the precision of the algorithm by giving more refined interpretations to a few special operators. These extensions make it possible to start experimenting with random interpretation for real programs, not just those restricted to linear computations.

7 Conclusion and Future Work

We have presented a global value numbering algorithm based on the idea of random interpretation. Our algorithm is perhaps the first polynomial time algorithm that discovers all equivalences in a program with non-deterministic conditionals and uninterpreted operators. An important feature of this algorithm is the simplicity of its data structures and of the operations it performs. Our algorithm does not require any symbolic manipulations like other global value numbering algorithms. We are working on proving better upper bounds on the value of k for probabilistic soundness; this can reduce the time complexity of our algorithm.

The next step will be to implement this algorithm and compare it with other existing algorithms on several benchmarks. We also plan to use the value numbering algorithm proposed in this paper as part of the translation validation infrastructure [10]. We hope that this will reduce several of the false alarms currently generated by the translation validation tool.

An interesting open problem is to discover all equivalences in a program with non-deterministic conditionals and expressions that involve both uninterpreted operators and linear arithmetic. Another interesting open problem is to consider the case when some of the uninterpreted operators are known to be commutative (for e.g. floating point arithmetic operators) or associative or both. We feel that randomization has much to offer to program analysis and this area is worthy of future research. Combining the randomized techniques with symbolic ones also seems to be a promising direction for future work.

8 References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11. ACM, 1988.

[2] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, June 1997.

[3] C. Click. Global code motion/global value numbering. In *Proccedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 246–257, June 1995.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1990.

[6] 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.

[7] S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *30th Annual ACM Symposium on Principles of Programming Languages*, pages 74–84. ACM, Jan. 2003.

[8] G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on Principles of Programming Language*, pages 194–206. ACM, Oct. 1973.

[9] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

[10] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 83–94. ACM SIGPLAN, 18–21 June 2000.

[11] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume LNCS 1384, pages 151–166. Springer, 1998.

[12] 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.

[13] O. Rüthing, J. Knoop, and B. Steffen. The value flow graph: A program representation for optimal program transformations. In N. D. Jones, editor, *Proceedings of the European Symposium on Programming*, pages 389–405. Springer-Verlag LNCS 432, 1990.

[14] O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 1999.

[15] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701–717, Oct. 1980.

[16] B. Steffen. Optimal run time optimization - proved by a new look at abstract interpretations. In *2nd International Joint Conference on Theory and Practice of Software Development (TAPSOFT'87)*, volume 249 of *LNCS*, pages 52–68. Springer-Verlag, March 1987.

[17] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

A Proof of Completeness and Soundness Theorems

We now give the proofs for the completeness and soundness theorems stated in Section 4.3.2. Both the abstract interpreter $\mathcal{A}$ and the random interpreter $\mathcal{R}$ perform similar operations for each node in the flow-graph. The proofs are by induction on the number of operations performed by the interpreters. The computation performed by the interpreters can be viewed as going forward in the sense that the outputs of a flowchart node are determined by the inputs of the node. Hence, for the inductive case of the proof, we prove that the required property holds for the outputs of the node given that it holds for the inputs of the node.

A.1 Proof of Completeness (Theorem 9)

The proof is by induction on the number of operations performed by the interpreters. The base case is trivial since initially $U = \emptyset$. Since $\emptyset \Rightarrow e_1 = e_2$, it must be the case that $e_1 = e_2$. Hence, $\tilde{S} \models e_1 = e_2$. For the inductive case, the following scenarios arise.

- Assignment Node: See Figure 4 (a).
  Consider the expressions $e'_1 = e_1[e/x]$ and $e'_2 = e_2[e/x]$. Since $U \Rightarrow e_1 = e_2$, $U' \Rightarrow e'_1 = e'_2$. It follows from the induction hypothesis on U' and $\tilde{S}'$ that $\tilde{S}' \models e'_1 = e'_2$. Hence, $\tilde{S} \models e_1 = e_2$.
- Conditional Node. See Figure 4 (b).
  This case is trivial since $U^1 = U^2 = U'$ and $\tilde{S}^1 = \tilde{S}^2 = \tilde{S}'$. By using the induction hypothesis on $\tilde{S}'$ and U', we get the desired result.
- Join Node: See Figure 4 (c).
  By definition of $\mathcal{A}$, $U^1 \Rightarrow e_1 = e_2$ and $U^2 \Rightarrow e_1 = e_2$. By induction hypothesis on U^1 and $\tilde{S}^1$ and on U^2 and $\tilde{S}^2$, we have that $\tilde{S}^1 \models e_1 = e_2$ and $\tilde{S}^2 \models e_1 = e_2$. It now follows from Lemma 10 that $\tilde{S} \models e_1 = e_2$.

A.2 Proof of Soundness (Theorem 11)

The proof is again by induction on the number of operations performed by the interpreters. For the base case, $V(e_1,k,\tilde{S}) = P(e_1,k)$ and $V(e_2,k,\tilde{S}) = P(e_2,k)$ since $\tilde{S}_k[x] = x$. Since an expression of depth t can have at most 2^t leaves when expressed as a tree, it follows from Lemma 7 that $e_1 = e_2$. Hence $U \Rightarrow e_1 = e_2$. For the inductive case, the following scenarios arise.

- See Figure 4 (a).
  Consider the expressions $e'_1 = e_1[e/x]$ and $e'_2 = e_2[e/x]$. Note that $\tilde{S}' \models e'_1 = e'_2$ since $\tilde{S} \models e_1 = e_2$. Also

note that $degree(V(e_1,k,\tilde{S})) = degree(V(e'_1,k,\tilde{S}'))$ and $degree(V(e_2,k,\tilde{S})) = degree(V(e'_2,k,\tilde{S}'))$. Hence, $k \geq min(degree(V(e'_1,k,\tilde{S}')), degree(V(e'_2,k,\tilde{S}')))$ since $k \geq min(degree(V(e_1,k,\tilde{S})), degree(V(e_2,k,\tilde{S})))$. It follows from the induction hypothesis on U', S', e'_1 and e'_2 that $U' \Rightarrow e'_1 = e'_2$. Thus, it follows that $U \Rightarrow e_1 = e_2$.

- See Figure 4 (b).
  This case is trivial since $\tilde{S}^1 = \tilde{S}^2 = \tilde{S}'$ and $U^1 = U^2 = U'$. The induction hypothesis on $\tilde{S}'$ and U' implies the desired result.

- See Figure 4 (c).
  By definition, $\tilde{V}(e_1,k,\tilde{S}) = w \times \tilde{V}(e_1,k,\tilde{S}^1) + (1-w) \times \tilde{V}(e_1,k,\tilde{S}^2)$ and $\tilde{V}(e_2,k,\tilde{S}) = w \times \tilde{V}(e_2,k,\tilde{S}^1) + (1-w) \times \tilde{V}(e_2,k,\tilde{S}^2)$, where w is a variable that does not occur in $\tilde{V}(e_1,k,\tilde{S}^1)$, $\tilde{V}(e_1,k,\tilde{S}^2)$, $\tilde{V}(e_2,k,\tilde{S}^1)$ or $\tilde{V}(e_2,k,\tilde{S}^2)$. Since $\tilde{V}(e_1,k,\tilde{S}) = \tilde{V}(e_2,k,\tilde{S})$, it follows that $\tilde{V}(e_1,k,\tilde{S}^1) = \tilde{V}(e_2,k,\tilde{S}^1)$ (by substituting $w = 0$). Hence, $\tilde{S}^1 \models e_1 = e_2$. Also, $degree(V(e_1,k,\tilde{S}^1)) \leq degree(V(e_1,k,\tilde{S})$ and $degree(V(e_2,k,\tilde{S}^1)) \leq degree(V(e_2,k,\tilde{S}))$. Thus, it follows from the induction hypothesis on $U^1, \tilde{S}^1, e_1$ and e_2 that $U^1 \Rightarrow e_1 = e_2$. Similarly, we can prove that $U^2 \Rightarrow e_1 = e_2$. It now follows from the definition of the abstract interpreter $\mathcal{A}$ that $U \Rightarrow e_1 = e_2$.

B Proof of Lemma 13

We first introduce some useful notation. We say that a pair $H = (I,E)$ has property P if $I \subseteq T$ and E is a set of equivalences $x = e$, one for each variable $x \in T - I$, such that $Vars(e) \subseteq I$ and $e \prec x$. For any pair $H = (I,E)$ with property P, let $\prec_E$ denote a partial order on the program variables such that $y \prec_E z$ iff $E \Rightarrow z = F(e_1,e_2)$ such that $y \in Vars(F(e_1,e_2))$, or $E \Rightarrow y = z$ such that $y \prec z$.

For any equivalence $e_1 = e_2$, let $D_{e_1=e_2}$ denote the following set of equivalences.

$$\begin{array}{rcl} D_{F(e'_1,e'_2)=F(e''_1,e''_2)} & = & D_{e'_1=e''_1} \cup D_{e'_2=e''_2} \\ D_{y=y} & = & \{\} \\ D_{y=e} & = & \{y = e\},\ \textit{where } e \not\equiv y \end{array}$$

Note that $D_{e_1=e_2}$ contains only equivalences of the form $y = e$, where y is some variable and e is some expression. Observe that

$$E \Rightarrow e_1 = e_2 \quad \text{iff} \quad E \Rightarrow D_{e_1=e_2}$$

(By $E \Rightarrow D_{e_1=e_2}$, we mean that $E \Rightarrow e = e'$ for every equivalence $e = e'$ in $D_{e_1=e_2}$.)

We now prove the lemma. The proof of the lemma is by induction on structure of the program. The base case is trivial since initially $H = (\emptyset, T)$. Clearly H has property P.

For the inductive case, the following scenarios arise.

- Assignment Node: See Figure 4 (a).
  Let $H' = (I',E')$ represent the set of Herbrand equivalences before the assignment node such that H' has property P. We prove that $H = (I,E)$, as defined below, has property P and it represents the set of Herbrand equivalences after the assignment node.

$$\begin{array}{rcl} E_t & = & E'[x'/x] \cup \{x = Sub(e,E')[x'/x]\} \\ I & = & \{y \mid y \in T, \neg\exists e'(E_t \Rightarrow y = e',\ e' \prec y,\ x' \notin Vars(e'))\} \\ E & = & \{y = e' \mid y \in T - I,\ E_t \Rightarrow y = e', e' \prec y, Vars(e') \subseteq I\} \end{array}$$

Here $Sub(e,E')$ refers to the expression obtained from e by replacing all occurrences of variable y in e by e' for all equivalences $y = e'$ in E'. Note that x' represents a fresh variable that does not occur in T.

It is not difficult to see that $H = (I,E)$ has property P. Clearly, E_t represents the set of Herbrand equivalences after the assignment node, since it is the strongest postcondition of E with respect to the assignment $x := e$. Hence, it suffices to show that if $E_t \Rightarrow e_1 = e_2$ such that $x' \notin Vars(e_1)$ and $x' \notin Vars(e_2)$, then $E \Rightarrow e_1 = e_2$. Suppose $E_t \Rightarrow e_1 = e_2$. Then, $E_t \Rightarrow D_{e_1=e_2}$. It now follows from Claim 16 stated and proved below that $E \Rightarrow D_{e_1=e_2}$. Hence, $E \Rightarrow e_1 = e_2$.

CLAIM 16. *Suppose $E_t \Rightarrow y = e'$ such that $x' \notin Vars(e')$ and $x' \not\equiv y$. Then $E \Rightarrow y = e'$.*

PROOF. Without loss of generality, we can assume that $e' \prec y$.

The proof is by induction on y with respect to the ordering $\prec_E$. The base case corresponds to y being in the set I, and hence $e' \equiv y$. Clearly, $E \Rightarrow y = y$. We now consider the inductive case. Since $E_t \Rightarrow y = e'$, there exists an expression e'' such that $(y = e'') \in E$. Note that $E_t \Rightarrow y = e''$ and hence $E_t \Rightarrow e' = e''$. Thus, $E_t \Rightarrow D_{e'=e''}$. Note that for every $z = e''' \in D_{e'=e''}$, $z \prec_E y$; and hence, it follows from the inductive hypothesis that $E \Rightarrow z = e'''$. Thus, $E \Rightarrow D_{e'=e''}$. Thus, $E \Rightarrow e' = e''$ and hence $E \Rightarrow y = e'$. □

- Conditional Node: See Figure 4 (b).
  This case is trivial since the sets of Herbrand equivalences that are true on the two branches are same as the set of Herbrand equivalences that are true before the conditional.

- Join Node. See Figure 4 (c).
  Let $H_1 = (I_1,E_1)$ and $H_2 = (I_2,E_2)$ represent the set of Herbrand equivalences before the join node such that both H_1 and H_2 have property P. We prove that $H = (I,E)$, as defined below, has property P and it represents the set of Herbrand equivalences after the join node.

$$\begin{array}{rcl} I & = & \{y \mid y \in T, \neg\exists e'(E_1 \Rightarrow y = e',\ E_2 \Rightarrow y = e', e' \prec y,)\} \\ E & = & \{y = e' \mid y \in T - I,\ E_1 \Rightarrow y = e',\ E_2 \Rightarrow y = e', \\ & & \qquad e' \prec y, Vars(e') \subseteq I\} \end{array}$$

It is not difficult to see that $H = (I,E)$ has property P. It suffices to show that if $E_1 \Rightarrow e_1 = e_2$ and $E_2 \Rightarrow e_1 = e_2$, then $E \Rightarrow e_1 = e_2$. Suppose $E_1 \Rightarrow e_1 = e_2$ and $E_2 \Rightarrow e_1 = e_2$. Then, $E_1 \Rightarrow D_{e_1=e_2}$ and $E_2 \Rightarrow D_{e_1=e_2}$. It now follows from Claim 17 stated and proved below that $E \Rightarrow D_{e_1=e_2}$. Hence, $E \Rightarrow e_1 = e_2$.

CLAIM 17. *Suppose $E_1 \Rightarrow y = e'$ and $E_2 \Rightarrow y = e'$. Then $E \Rightarrow y = e'$.*

PROOF. Without loss of generality, we can assume that $e' \prec y$. The proof is by induction on y with respect to the ordering $\prec_{E_1}$. The base case corresponds to y being in the set I_1, and hence $e' \equiv y$. Clearly, $E \Rightarrow y = y$. We now consider the inductive case. Since $E_1 \Rightarrow y = e'$ and $E_2 \Rightarrow y = e'$, there exists an expression e'' such that $(y = e'') \in E$. Since $E \Rightarrow y = e''$, note that $E_1 \Rightarrow y = e''$ and $E_2 \Rightarrow y = e''$. Hence, $E_1 \Rightarrow e' = e''$ and $E_2 \Rightarrow e' = e''$. Thus, $E_1 \Rightarrow D_{e'=e''}$ and $E_2 \Rightarrow D_{e'=e''}$. For every $z = e''' \in D_{e'=e''}$, $z \preceq_{E_1} y$; and hence, it follows from the inductive hypothesis that $E \Rightarrow (z = e''')$. Thus, $E \Rightarrow D_{e'=e''}$. Thus, $E \Rightarrow e' = e''$ and hence $E \Rightarrow y = e'$. □