Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic

Randal E. Bryant, Steven German,[1] Miroslav N. Velev

May, 1999
CMU-CS-99-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The CMU authors were supported by the Semiconductor Research Corporation under contract 98-DC-068, as well as by grants from Motorola, Intel, and Fujitsu.

[1]IBM Watson Research Center

Keywords: Formal verification, Processor verification, Uninterpreted functions, Decision procedures

Abstract

The logic of equality with uninterpreted functions (EUF) provides a means of abstracting the manipulation of data by a processor when verifying the correctness of its control logic. By reducing formulas in this logic to propositional formulas, we can apply Boolean methods such as Ordered Binary Decision Diagrams (BDDs) and Boolean satisfiability checkers to perform the verification. We can exploit characteristics of the formulas describing the verification conditions to greatly simplify the propositional formulas generated. We identify a class of terms we call “p-terms” for which equality comparisons can only be used in monotonically positive formulas. By applying suitable abstractions to the hardware model, we can express the functionality of data values and instruction addresses flowing through an instruction pipeline with p-terms. A decision procedure can exploit the restricted uses of p-terms by considering only “maximally diverse” interpretations of the associated function symbols, where every function application yields a different value except when constrained by functional consistency.

We present two methods to translate formulas in EUF into propositional logic. The first interprets the formula over a domain of fixed-length bit vectors and uses vectors of propositional variables to encode domain variables. The second generates formulas encoding the conditions under which pairs of terms have equal valuations, introducing propositional variables to encode the equality relations between pairs of terms. Both of these approaches can exploit maximal diversity to greatly reduce the number of propositional variables that need to be introduced and to reduce the overall formula sizes.

We present experimental results demonstrating the efficiency of this approach when verifying pipelined processors using the method proposed by Burch and Dill. Exploiting positive equality allows us to overcome the exponential blow-up experienced previously [VB98] when verifying microprocessors with load, store, and branch instructions.

1 Introduction

For automatically reasoning about pipelined processors, Burch and Dill demonstrated the value of using propositional logic, extended with uninterpreted functions, uninterpreted predicates, and the testing of equality [BD94]. Their approach involves abstracting the data path as a collection of registers and memories storing data, units such as ALUs operating on the data, and various connections and multiplexors providing methods for data to be transferred and selected. The initial state of each register is represented by a domain variable indicating an arbitrary data value. The operation of units that transform data is abstracted as blocks computing functions with no specified properties other than functional consistency, i.e., that applications of a function to equal arguments yield equal results: $x = y \Rightarrow f(x) = f(y)$. The state of a register at any point in the computation can be represented by a symbolic term, an expression consisting of a combination of domain variables, function and predicate applications, and Boolean operations. Verifying that a pipelined processor has behavior matching that of an unpipelined instruction set reference model can be performed by constructing a formula in this logic that compares for equality the terms describing the results produced by the two models and then proving the validity of this formula.

In their 1994 paper, Burch and Dill also described the implementation of a decision procedure for this logic based on theorem proving search methods. Their procedure builds on ones originally described by Shostak [Sho79] and by Nelson and Oppen [NO80], using combinatorial search coupled with algorithms for maintaining a partitioning of the terms into equivalence classes based on the equalities that hold at a given step of the search. More details of their decision procedure are given in [BDL96].

Burch and Dill's work has generated considerable interest in the use of uninterpreted functions to abstract data operations in processor verification. A common theme has been to adopt Boolean methods, either to allow integration of uninterpreted functions into symbolic model checkers [DPR98, BBCZ98], or to allow the use of Binary Decision Diagrams (BDDs) [Bry86] in the decision procedure [HKGB97, GSZAS98, VB98]. Boolean methods allow a more direct modeling of the control logic of hardware designs and thus can be applied to actual processor designs rather than highly abstracted models. In addition to BDD-based decision procedures, Boolean methods could use some of the recently developed satisfiability procedures for propositional logic. In principle, Boolean methods could outperform decision procedures based on theorem proving search methods, especially when verifying processors with more complex control logic, e.g., due to superscalar or out-of-order operation.

Boolean methods can be used to decide the validity of a formula containing terms and uninterpreted functions by interpreting the formula over a domain of fixed-length bit vectors. Such an approach exploits the property that a given formula contains a limited number of function applications and therefore can be proved to be universally valid by considering its interpretation over a sufficiently large, but finite domain [Ack54]. If a formula contains a total of m function applications, then the set of all bit vectors of length k forms an adequate domain for $k \geq \log_2 m$. The formula to be verified can be translated into one in propositional logic, using vectors of propositional variables to encode the possible values generated by function applications [HKGB97]. Our implementation of such an approach [VB98] as part of a BDD-based symbolic simulation system was successful at verifying simple pipelined data paths. We found, however, that the computational resources grew exponentially as we increased the pipeline depth. Modeling the interactions between successive instructions flowing through the pipeline, as well as the functional consistency of the ALU results, precludes having an ordering of the variables encoding term values that yields compact BDDs. Similarly, we found that extending the data path to a complete processor by adding either load and store instructions or instruction fetch logic supporting jumps and conditional branches led to impossible BDD variable ordering requirements.

Goel *et al.* [GSZAS98] present an alternate approach to using BDDs to decide the validity of formulas in the logic of equality with uninterpreted functions. In their formulation they introduce a propositional variable $e_{i,j}$ for each pair of function application terms T_i and T_j, expressing the conditions under which the two terms are equal. They add constraints expressing both functional consistency and the transitivity of equality among the terms. Their experimental results were also somewhat disappointing. For all previous methods of reducing EUF to propositional logic, Boolean methods have not lived up to their promise of outperforming ones based on theorem proving search.

In this paper, we show that the characteristics of the formulas generated when modeling processor pipelines can be exploited to greatly reduce the number of propositional variables that are introduced when translating the formula into propositional logic. We distinguish a class of terms we call *p-terms* for which equality comparisons can only be used in monotonically positive formulas. Such formulas are suitable for describing the top-level correctness condition, but not for modeling any control decisions in the hardware. By applying suitable abstractions to the hardware model, we can express the functionality of data values and instruction addresses with p-terms.

A decision procedure can exploit the restricted uses of p-terms by considering only "maximally diverse" interpretations of the associated "p-function" symbols, where every function application yields a different value except when constrained by functional consistency. We present a method of transforming a formula containing function applications into one containing only domain variables that differs from the commonly-used method described by Ackermann [Ack54]. Our method allows a translation into propositional logic that uses vectors with fixed bit patterns rather than propositional variables to encode domain variables introduced while eliminating p-function applications. This reduction in propositional variables greatly simplifies the BDDs generated when checking tautology, often avoiding the exponential blow-up experienced by other procedures. Alternatively, we can use a encoding scheme similar to Goel *et al.* [GSZAS98], but with many of the $e_{i,j}$ values set to $\mathbf{false}$ rather than to Boolean variables.

Others have recognized the value of restricting the testing of equality when modeling the flow of data in pipelines. Berezin *et al.* [BBCZ98] generate a model of an execution unit suitable for symbolic model checking in which the data values and operations are kept abstract. In our terminology, their functional terms are all p-terms. They use fixed bit patterns to represent the initial states of registers, much as we replace p-term domain variables by fixed bit patterns. To model the outcome of each program operation, they generate an entry in a "reference file" and refer to the result by a pointer to this file. These pointers are similar to the bit patterns we generate to denote the p-function application outcomes. This paper provides an alternate, and somewhat more general view of the efficiency gains allowed by p-terms.

Damm *et al.* consider an even more restricted logic such that in the terms describing the computed result, no function symbol is applied to a term that already contains the same symbol. As a consequence, they can guarantee that an equality between two terms holds universally if it holds holds over the domain $\{0, 1\}$ and with function symbols having four possible interpretations: constant functions 0 or 1, and projection functions selecting the first or second argument. They can therefore argue that verifying an execution unit in which the data path width is reduced to a single bit and in which the functional units implement only four functions suffices to prove its correctness for all possible widths and functionalities. Their work imposes far greater restrictions than we place on p-terms, but it allows them to bound the domain that must be considered to determine universal validity independently from the formula size.

In comparison to both of these other efforts, we maintain the full generality of the unrestricted terms of Burch and Dill while exploiting the efficiency gains possible with p-terms. In our processor model, we can abstract register identifiers as unrestricted terms, while modeling program data and instruction data as p-terms. As a result, our verifications cover designs with arbitrarily many registers. In contrast, both [BBCZ98] and [DPR98] used bit encodings of register identifiers and were unable to scale their verifications to a realistic number of registers.

In a recent paper, Pnueli, *et al.* [PRSS99] also propose a method to exploit the polarity of the equations in a formula containing uninterpreted functions with equality. They describe an algorithm to generate a small domain for each domain variable such that the universal validity of the formula can be determined by considering only interpretations in which the variables range over their restricted domains. A key difference of their work is that they examine the equation structure after replacing all function application terms with domain variables and introducing functional consistency constraints as described by Ackermann [Ack54]. These consistency constraints typically contain large numbers of equations—far more than occur in the original formula—that mask the original p-term structure. As an example, comparing the top and bottom parts of Figure 6 illustrates the large number of equations that may be generated when applying Ackermann's method. By contrast, our method is based on the original formula structure. In addition, we use a new method of replacing function application terms with domain variables. Our scheme allows us to exploit maximal diversity by assigning fixed values to the domain variables generated while expanding p-function application terms.

The remainder of the paper is organized as follows. We define the syntax and semantics of our logic by extending that of Burch and Dill's. We prove our central result concerning the need to consider only maximally diverse interpretations when deciding the validity of formulas in our logic. As a first step in transforming our logic into propositional logic, we describe a new method of eliminating function application terms in a formula. Building on this, we describe two methods of translating formulas into propositional logic and show how these methods can exploit the

$$
\begin{array}{rcl}
\textit{term} & ::= & \textit{ITE}(\textit{formula}, \textit{term}, \textit{term}) \\
 & & \mid \textit{function-symbol}(\textit{term}, \ldots, \textit{term}) \\
 & & \\
\textit{formula} & ::= & \mathbf{true} \mid \mathbf{false} \mid \neg\textit{formula} \\
 & & \mid (\textit{formula} \wedge \textit{formula}) \mid (\textit{formula} \vee \textit{formula}) \\
 & & \mid (\textit{term} = \textit{term}) \\
 & & \mid \textit{predicate-symbol}(\textit{term}, \ldots, \textit{term})
\end{array}
$$

Figure 1: Syntax Rules for the Logic of Equality with Uninterpreted Functions (EUF)

properties of p-terms. We discuss the abstractions required to model processor pipelines in our logic. Finally, we present experimental results showing our ability to verify a simple, but complete pipelined processor.

2 Logic of Equality with Uninterpreted Functions (EUF)

The logic of Equality with Uninterpreted Functions (EUF) presented by Burch and Dill [BD94] can be expressed by the syntax given in Figure 1. In this logic, *formulas* have truth values while *terms* have values from some arbitrary domain. Terms are formed by application of uninterpreted function symbols and by applications of the *ITE* (for "if-then-else") operator. The *ITE* operator chooses between two terms based on a Boolean control value, i.e., *ITE*$(\mathbf{true}, x_1, x_2)$ yields x_1 while *ITE*$(\mathbf{false}, x_1, x_2)$ yields x_2. Formulas are formed by comparing two terms with equality, by applying an uninterpreted predicate symbol to a list of terms, and by combining formulas using Boolean connectives. A formula expressing equality between two terms is called an *equation*. We use *expression* to refer to either a term or a formula.

Every function symbol f has an associated *order*, denoted $ord(f)$, indicating the number of terms it takes as arguments. Function symbols of order zero are referred to as *domain variables*. We use the shortened form v rather than $v()$ to denote an instance of a domain variable. Similarly, every predicate p has an associated order $ord(p)$. Predicates of order zero are referred to as *propositional variables*, and can be written a rather than $a()$.

The truth of a formula is defined relative to a nonempty domain $\mathcal{D}$ of values and an interpretation I of the function and predicate symbols. Interpretation I assigns to each function symbol of order k a function from $\mathcal{D}^k$ to $\mathcal{D}$, and to each predicate symbol of order k a function from $\mathcal{D}^k$ to $\{\mathbf{true}, \mathbf{false}\}$. For the special case of order 0 symbols, i.e., domain (respectively, propositional) variables, the interpretation assigns an element of $\mathcal{D}$ (resp., $\{\mathbf{true}, \mathbf{false}\}$.) Given an interpretation I of the function and predicate symbols and an expression E, we can define the *valuation* of

Form E	Valuation $I[E]$
true	**true**
false	**false**
$\neg F$	$\neg I[F]$
$F_1 \wedge F_2$	$I[F_1] \wedge I[F_2]$
$p(T_1, \ldots, T_k)$	$I(p)(I[T_1], \ldots, I[T_k])$
$T_1 = T_2$	$I[T_1] = I[T_2]$
ITE(F, T_1, T_2)	*ITE*$(I[F], I[T_1], I[T_2])$
$f(T_1, \ldots, T_k)$	$I(f)(I[T_1], \ldots, I[T_k])$

Table 1: Evaluation of EUF Formulas and Terms

E under I, denoted $I[E]$, according to its syntactic structure. The valuation is defined recursively, as shown in Table 1. $I[E]$ will be an element of the domain when E is a term, and a truth value when E is a formula.

A formula F is said to be *true under interpretation* I when $I[F] = \texttt{true}$. It is said to be *valid over domain* $\mathcal{D}$ when it is true over domain $\mathcal{D}$ for all interpretations of the symbols in F. F is said to be *universally valid* when it is valid over all domains. A basic property of validity is that a given formula is valid over a domain $\mathcal{D}$ iff it is valid over all domains having the same cardinality as $\mathcal{D}$. This follows from the fact that a given formula has the same truth value in any two isomorphic interpretations of the symbols in the formula. Another property of the logic, which can be readily shown, is that if F is valid over a suitably large domain, then it is universally valid [Ack54]. In particular, it suffices to have a domain as large as the number of syntactically distinct function application terms occurring in F. We are interested in decision procedures that determine whether or not a formula is universally valid; we will show how to do this by dynamically constructing a sufficiently large domain as the formula is being analyzed.

3 Positive Equality with Uninterpreted Functions (PEUF)

We can improve the efficiency of validity checking by treating positive and negative equations differently when reducing EUF to propositional logic. Informally, an equation is positive if it does not appear negated in a formula. In particular, a positive equation cannot appear as the formula that controls the value of an *ITE* term; such formulas are considered to appear both positively and negatively.

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

Figure 2: Syntax Rules for the Logic of Positive Equality with Uninterpreted Functions (PEUF)

3.1 Syntax

PEUF is an extended logic based on EUF; its syntax is shown in Figure 2. The main idea is that there are two disjoint classes of function symbols, called p-function symbols and g-function symbols, and two classes of terms.

General terms, or *g-terms*, correspond to terms in EUF. Syntactically, a g-term is a g-function application or an *ITE* term in which the two result terms are hereditarily built from g-function applications and *ITE*s.

The new class of terms is called positive terms, or *p-terms*. P-terms may not appear in negated equations, i.e., equations within the scope of a logical negation. Since p-terms can contain p-function symbols, the syntax is restricted in a way that prevents p-terms from appearing in negative equations. When two p-terms are compared for equality, the result is a special, restricted kind of formula called a *p-formula*.

Note that our syntax allows any g-term to be "promoted" to a p-term. Throughout the syntax definition, we require function and predicate symbols to take p-terms as arguments. However, since g-terms can be promoted, the requirement to use p-terms as arguments does not restrict the use of g-function symbols or g-terms. In essence, g-function symbols may be used as freely in our logic as in EUF, but the p-function symbols are restricted. To maintain the restriction on p-function symbols, the syntax does not permit a p-term to be promoted to a g-term.

A *formula* of the extended logic is a Boolean combination of equations on g-terms and applications of predicate symbols. Formulas in our logic serve as Boolean control expressions in *ITE* terms. A formula can contain negation, and *ITE* implicitly negates its Boolean control, so only g-terms are allowed in equations in formulas. Since a predicate formula $p(T_1, \ldots, T_k)$, where p is a predicate symbol and the T_i are terms, is not an equation, we allow the terms in predicate formulas to be chosen from the largest class of terms, namely the p-terms.

Finally, the syntactic class *p-formula* is the class for which we develop validity checking methods. P-formulas are built up using only the monotonically positive Boolean operations $\wedge$ and $\vee$. P-formulas may not be placed under a negation sign, and cannot be used as the control for an *ITE* operation. As described in later sections, our validity checking methods will take advantage of the assumption that in p-formulas, the p-terms cannot appear in negative equations.

Observe that PEUF does not extend the expressive power of EUF—we could translate any PEUF expression into EUF by considering the g-terms and p-terms to be terms and the p-formulas to be formulas. Instead, the benefit of PEUF is that by distinguishing some portion of a formula as satisfying a restricted set of properties, we can radically reduce the number of different interpretations we must consider when proving that a p-formula is universally valid.

As a running example for this paper, we consider the formula $x = y \Rightarrow h(g(x), g(g(x))) = h(g(y), g(g(x)))$, which would be transformed into a p-formula F_{eg} by eliminating the implication:

$$F_{\mathrm{eg}} \quad = \quad \neg(x = y) \quad \vee \quad h(g(x), g(g(x))) = h(g(y), g(g(x))) \tag{1}$$

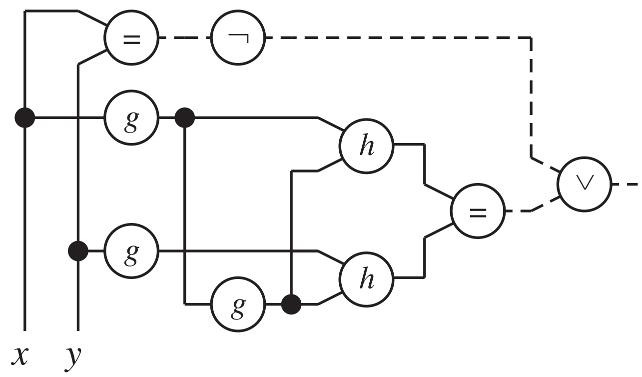


Figure 3: Schematic Representation of F_{eg}. Domain values are shown as solid lines, while truth values are shown as dashed lines.

Domain variables x and y must be g-function symbols so that we can consider the equation $x=y$ to be a formula, and hence it can be negated to give formula $\neg(x=y)$. We can promote the g-terms x and y to p-terms, and we can consider function symbols g and h to be p-function symbols, giving p-terms $g(x)$, $g(y)$, $g(g(x))$, $h(g(x), g(g(x)))$, and $h(g(y), g(g(x)))$. Thus, the equation $h(g(x), g(g(x))) = h(g(y), g(g(x)))$ is a p-formula. We form the disjunction of this p-formula with the p-formula obtained by promoting $\neg(x=y)$ giving p-formula F_{eg}.

Figure 3 shows a schematic representation of F_{eg}, using drawing conventions similar to those found in hardware designs. That is, we view domain variables as inputs (shown along bottom) to a network of operators. Domain values are denoted with solid lines, while truth values are denoted with dashed lines. The top-level formula then becomes the network output, shown on the right. The operators in the network are shared whenever possible. This representation is isomorphic to the traditional directed acyclic graph (DAG) representation of an expression, with maximal sharing of common subexpressions.

3.2 Diverse Interpretations

Let $\mathcal{T}$ be a set of terms, where a term may be either a g-term or a p-term. We consider two terms to be distinct only if they differ syntactically. An expression may therefore contain multiple instances of a single term. We classify terms as either p-function applications, g-function applications, or *ITE* terms, according to their top-level operation. The first two categories are collectively referred to as function application terms. For any formula or p-formula F, define $\mathcal{T}(F)$ as the set of all function application terms occurring in F.

An interpretation I partitions a term set $\mathcal{T}$ into a set of equivalence classes, where terms T_1 and T_2 are equivalent under I, written $T_1 \approx_I T_2$ when $I[T_1] = I[T_2]$. Interpretation I' is said to be a *refinement* of I for term set $\mathcal{T}$ when $T_1 \approx_{I'} T_2 \Rightarrow T_1 \approx_I T_2$ for every pair of terms T_1 and T_2 in $\mathcal{T}$. I' is a *proper* refinement of I for $\mathcal{T}$ when it is a refinement and there is at least one pair of

I1	$\{x,y\},\{g_1\}\{g_2\},\{g_3\},\{h_1\},\{h_2\}$	Inconsistent
I2	$\{x\},\{y\},\{g_1,g_2\},\{g_3\},\{h_1\},\{h_2\}$	Inconsistent
C1	$\{x\},\{y\},\{g_1,g_2\},\{g_3\},\{h_1,h_2\}$	Diverse w.r.t. x,y,h
C2	$\{x,g_3\},\{y\},\{g_1\},\{g_2\},\{h_1\},\{h_2\}$	Diverse w.r.t. y, h
D1	$\{x\},\{y\},\{g_1\},\{g_2\},\{g_3\},\{h_1\},\{h_2\}$	Diverse w.r.t. x, y, g, h
D2	$\{x,y\},\{g_1,g_2\},\{g_3\},\{h_1,h_2\}$	Diverse w.r.t. g, h

Table 2: Example Partitionings of Terms x, y, $g_1 \doteq g(x)$, $g_2 \doteq g(y)$, $g_3 \doteq g(g(x))$, $h_1 \doteq h(g(x), g(g(x)))$, and $h_2 \doteq h(g(y), g(g(x)))$.

terms $T_1, T_2 \in \mathcal{T}$ such that $T_1 \approx_I T_2$, but $T_1 \not\approx_{I'} T_2$.

Let Σ denote a subset of the function symbols in formula F. An interpretation I is said to be *diverse* for F with respect to Σ when it provides a maximal partitioning of the function application terms in $\mathcal{T}(F)$ having a top-level function symbol from Σ relative to each other and to the other function application terms, but subject to the constraints of functional consistency. That is, for T_1 of the form $f(T_{1,1}, \ldots, T_{1,k})$, where $f \in \Sigma$, an interpretation I is diverse with respect to Σ if I has $T_1 \approx_I T_2$ only in the case where T_2 is also a term of the form $f(T_{2,1}, \ldots, T_{2,k})$, and $T_{1,i} \approx_I T_{2,i}$ for all i such that $1 \le i \le k$. If we let $\Sigma_p(F)$ denote the set of all p-function symbols in F, then interpretation I is said to be *maximally diverse* when it is diverse with respect to $\Sigma_p(F)$. Note that this property requires the p-function application terms to be in separate equivalence classes from the g-function application terms.

As an example, consider the p-formula F_{eg} given in Equation 1. There are seven distinct function application terms identified as follows:

x	y	g_1	g_2	g_3	h_1	h_2
x	y	$g(x)$	$g(y)$	$g(g(x))$	$h(g(x), g(g(x)))$	$h(g(y), g(g(x)))$

Table 2 shows 6 of the 877 different ways to partition seven objects into equivalence classes. Many of these violate functional consistency. For example, the partitioning I1 describes a case where x and y are equal, but $g(x)$ and $g(y)$ are not. Similarly, partitioning I2 describes a case where $g(x)$ and $g(y)$ are equal, but $h(g(x), g(g(x)))$ and $h(g(y), g(g(x)))$ are not.

Eliminating the inconsistent cases gives 384 partitionings. Many of these do not arise from maximally diverse interpretations, however. For example, partitioning C1 arises from an interpretation that is not diverse with respect to g, while partitioning C2 arises from an interpretation that is not diverse with respect to h. In fact, there are only two partitionings: D1 and D2 that arise from maximally diverse interpretations. Partition D1 corresponds to an interpretation that is diverse with respect to all of its function symbols. Partition D2 is diverse with respect to both g and h, even though terms g_1 and g_2 are in the same class, as are h_1 and h_2. Both of these groupings are forced by functional consistency: having $x = y$ forces $g(x) = g(y)$, which in turn forces $h(g(x), g(g(x))) = h(g(y), g(g(x)))$. Since g and h are the only p-function symbols, D2 is maximally diverse.

Theorem 1 *A p-formula F is universally valid if and only if it is true in all maximally diverse interpretations.*

First, it is clear that if F is universally valid, F is true in all maximally diverse interpretations. We prove via the following two lemmas that if F is true in all maximally diverse interpretations it is universally valid.

Lemma 1 *If interpretation J is not maximally diverse for p-formula F, then there is an interpretation J' that is a proper refinement of J such that $J'[F] \Rightarrow J[F]$.*

Proof: Let T_1 be a term occurring in F of the form $f_1(T_{1,1}, \ldots, T_{1,k_1})$, where f_1 is a p-function symbol. Let T_2 be a term occurring in F of the form $f_2(T_{2,1}, \ldots, T_{2,k_2})$, where f_2 may be either a p-function or a g-function symbol. Assume furthermore that $J[T_1]$ and $J[T_2]$ both equal z, but that either symbols f_1 and f_2 differ, or $J[T_{1,i}] \neq J[T_{2,i}]$ for some value of i.

Let z' be a value not in $\mathcal{D}$, and define a new domain $\mathcal{D}' \doteq \mathcal{D} \cup \{z'\}$. Our strategy is to construct an interpretation J' over $\mathcal{D}'$ that partitions the terms in $\mathcal{T}(F)$ in the same way as J, except that it splits the class containing terms T_1 and T_2 into two parts—one containing T_1 and evaluating to z', and the other containing T_2 and evaluating to z.

Define function $h\colon \mathcal{D}' \rightarrow \mathcal{D}$ to map elements of $\mathcal{D}'$ back to their counterparts in $\mathcal{D}$, i.e., $h(z') = z$, while all other values of x give $h(x)$ equal to x.

For p-function symbol f_1, define $J'(f_1)$ as:

$$J'(f_1)(x_1, \ldots, x_{k_1}) \quad \doteq \quad \begin{cases} z', & h(x_i) = J[T_{1,i}],\ 1 \leq i \leq k_1 \\ J(f_1)(h(x_1), \ldots, h(x_{k_1})), & \text{otherwise} \end{cases}$$

For other function and predicate symbols, J' is defined to preserve the functionality of interpretation J, while also treating argument values of z' the same as z. That is, $J'(f)$ for function symbol f having $ord(f)$ equal to k is defined such that $J'(f)(x_1, \ldots, x_k) = J(f)(h(x_1), \ldots, h(x_k))$. Similarly, $J'(p)$ for predicate symbol p having $ord(p)$ equal to k is defined such that $J'(p)(x_1, \ldots, x_k) = J(p)(h(x_1), \ldots, h(x_k))$.

We claim the following properties for the different forms of subexpressions occurring in F:

1. For every formula G: $J'[G] = J[G]$
2. For every g-term T: $J'[T] = J[T]$
3. For every p-term T: $h(J'[T]) = J[T]$

4. For every p-formula G: $J'[G] \Rightarrow J[G]$

5. $J'[T_1] = z'$ and $J'[T_2] = z$.

These properties can be proved by induction on the expression depths. Informally, interpretation J' maintains the values of all g-terms and formulas as occur under interpretation J. It also maintains the values of all p-terms, except those in the class containing terms T_1 and T_2. These p-terms are split into some having valuation z and others having valuation z'. With respect to p-formulas, consider first an equation of the form $S_1 = S_2$ where S_1 and S_2 are p-terms. The equation will yield the same value under both interpretations except under the condition that S_1 and S_2 are split into different parts of the class that originally evaluated to z, in which case the equation will yield $\mathtt{true}$ under J, but $\mathtt{false}$ under J'. Thus, although this equation can yield different values under the two interpretations, we always have that $J'[S_1 = S_2] \Rightarrow J[S_1 = S_2]$. This implication relation is preserved by conjunctions and disjunctions of p-formulas, due to the monotonicity of these operations.

We will now present this argument formally. We define the depth of an expression E, $depth(E)$, in the familiar way:

1. $depth(\mathbf{true}) = depth(\mathbf{false}) = 0$.

2. $depth(v) = 0$, for domain variable v.

3. $depth(a) = 0$, for propositional variable a.

4. For any other expression E, $depth(E)$ is given by 1 plus the maximum depth of a subexpression in E.

We prove hypotheses 1 to 4 above by simultaneous induction on the depth of expressions:

For the base case of the induction, we have:

1. Formula: $J'[\mathbf{true}] = J[\mathbf{true}]$, $J'[\mathbf{false}] = J[\mathbf{false}]$, and $J'[a] = J[a]$ for any propositional variable a.

2. G-term: If v a g-function symbol of zero order, then $J'(v) = J(v)$.

3. P-term: If v is a p-function symbol of zero order, then by the definition of J', $h(J'(v)) = J(v)$.

4. P-formula: same as formula.

For the induction case, we assume that the inductive hypotheses 1 through 4 hold for all expressions of depth $\leq n$, and show that the hypotheses hold for expressions of depth $n + 1$.

1. Formula: There are several cases, depending on the form of G.

   (a) Suppose G has one of the forms $\neg G_1, G_1 \wedge G_2, G_1 \vee G_2$, where G_1 and G_2 are formulas. By the inductive hypothesis, $J'[G_1] = J[G_1]$, and $J'[G_2] = J[G_2]$. It follows that $J'[\neg G_1] = J[\neg G_1]$, $J'[G_1 \wedge G_2] = J[G_1 \wedge G_2]$, and $J'[G_1 \vee G_2] = J[G_1 \vee G_2]$.

   (b) Suppose G has the form $S_1 = S_2$, where S_1, S_2 are g-terms. By the inductive hypothesis on g-terms, $J'[S_1] = J[S_1]$, and $J'[S_2] = J[S_2]$. It follows that $J'[S_1 = S_2] = J[S_1 = S_2]$.

   (c) The remaining case is that G is a predicate application of the form $p(S_1, \ldots, S_k)$, where p is a predicate symbol of order k, and $S_1, \ldots, S_k$, are p-terms. By the inductive hypothesis for p-terms, we have $h(J'[S_i]) = J[S_i]$, for $i = 1 \ldots k$. By the definition of J',

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

2. G-term: There are two cases.

   (a) Suppose T has the form $\mathit{ITE}(G, S_1, S_2)$, where G is a formula, and S_1 and S_2 are g-terms. By the inductive hypothesis, we have $J'[G] = J[G]$, $J'[S_1] = J[S_1]$, and $J'[S_2] = J[S_2]$. Then $J'[\mathit{ITE}(G, S_1, S_2)] = J[\mathit{ITE}(G, S_1, S_2)]$.

   (b) Suppose T has the form $f(S_1, \ldots, S_k)$, where f is a g-function symbol of order k and $S_1, \ldots, S_k$ are p-terms. By the inductive hypothesis, $h(J'[S_i]) = J[S_i]$, for $i = 1, \ldots, k$. Then we have,

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

3. P-term: There are three cases.

   (a) Suppose T is a g-term. By the inductive hypothesis, $J'[T] = J[T]$. Since $J[T]$ cannot be equal to z', it must be the case that $h(J'[T]) = J[T]$.

   (b) Suppose T has the form $\mathit{ITE}(G, S_1, S_2)$, where G is a formula, and S_1 and S_2 are p-terms. By the inductive hypothesis, $J'[G] = J[G]$, $h(J'[S_1]) = J[S_1]$, and $h(J'[S_2] = J[S_2])$. It follows that

$$\begin{aligned}
h(J'[\mathit{ITE}(G, S_1, S_2)]) &= \text{if } J'[G] \text{ then } h(J'[S_1]) \text{ else } h(J'[S_2]) \\
&= \text{if } J[G] \text{ then } J[S_1] \text{ else } J[S_2] \\
&= J[\mathit{ITE}(G, S_1, S_2)].
\end{aligned}$$

(c) Suppose that T has the form $f(S_1, \ldots, S_k)$, where f is a p-function symbol of order k and $S_1, \ldots, S_k$ are p-terms. Here, we have to consider two cases. The first case is that the following two conditions hold: (1) f is the function symbol f_1, i.e., the function symbol of the term T_1 mentioned at the beginning of the proof of this lemma, and (2) $h(S_i) = J[T_{1,i}]$, for $1 \leq i \leq k$. If these two conditions hold, then by the definition of J', $J'[f_1(S_1, \ldots, S_k)] = z'$, while $J[f_1(S_1, \ldots, S_k)] = z$. Since $h(z') = z$, we have $h(J'[f_1(S_1, \ldots, S_k)]) = J[f_1(S_1, \ldots, S_k)]$.

Now we consider the case that one of the two conditions mentioned above does not hold. The proof of this case is identical to the proof of case 2(b) above.

4. P-formula: There are three cases.

(a) If the p-formula G is a formula, then by the inductive hypothesis, $J'[G] = J[G]$, so $J'[G] \Rightarrow J[G]$.

(b) Suppose G has one of the forms $G_1 \wedge G_2$, or $G_1 \vee G_2$, where G_1, G_2 are p-formulas. By the inductive hypothesis, $J'[G_1] \Rightarrow J[G_1]$, and $J'[G_2] \Rightarrow J[G_2]$. Thus we have

$$\begin{aligned} J'[G_1 \wedge G_2] &= J'[G_1] \wedge J'[G_2] \\ &\Rightarrow J[G_1] \wedge J[G_2] \\ &= J[G_1 \wedge G_2], \end{aligned}$$

so $J'[G_1 \wedge G_2] \Rightarrow J[G_1 \wedge G_2]$. The proof for $G_1 \vee G_2$ is the same.

(c) Finally, we consider the case that G is a p-formula of the form $S_1 = S_2$, where S_1 and S_2, are p-terms. By the inductive hypothesis, we have that if $J'[S_i] = z'$, then $J[S_i] = z$, for $i = 1, 2$. Also, by the definition of h, we have that if $J'[S_i]$ does not equal z', then $J'[S_i] = J[S_i]$. Now, we consider cases depending on whether $J'[S_1]$ or $J'[S_2]$ are equal to z'. If both terms are equal to z' in J', then both $J[S_1]$ and $J[S_2]$ must be equal to z, so the equation is true in both J' and J. If neither $J'[S_1]$ nor $J'[S_2]$ is equal to z', then $J'[S_1] = J[S_1]$ and $J'[S_2] = J[S_2]$, so the equation has the same truth value in J' and J. The last case is that exactly one of the p-terms is equal to z' in J'. In this case, the equation is false in J', so we have $J'[G] \Rightarrow J[G]$. This completes the inductive proof.

Property 5 above, which implies that J' is a proper refinement, is a consequence of the definition of J' and the inductive properties 2 and 3. First, we show that $J'[T_1] = z'$. By definition, $J'[T_1] = J'(f_1)(J'[T_{1,1}], \ldots, J'[T_{1,k_1}])$. By property 3 on p-terms, we can assume $h(J'[T_{1,i}]) = J[T_{1,i}]$, for all i in the range $1 \leq i \leq k_1$. By the definition of $J'(f_1)$, we have $J'(f_1)(J'[T_{1,1}], \ldots, J'[T_{1,k_1}]) = z'$.

The proof that $J'[T_2] = z$ is in two cases, depending on whether T_1 and T_2 are applications of the same function symbol.

1. First, consider the case that $T_1 = f_1(T_{1,1}, \ldots, T_{1,k_1})$ and $T_2 = f_2(T_{2,1}, \ldots, T_{2,k_2})$, where f_1 and f_2 are different function symbols. In this case,

$$\begin{array}{rcl} J'[T_2] & = & J'(f_2)(J'[T_{2,1}], \ldots, J'[T_{2,k_2}]) \\ & = & J(f_2)(h(J'[T_{2,1}]), \ldots, h(J'[T_{2,k_2}])), \text{by the definition of } J'(f_2) \\ & = & J(f_2)(J[T_{2,1}], \ldots, J[T_{2,k_2}]), \text{by the inductive hypothesis} \\ & = & J[f_2(T_{2,1}, \ldots, T_{2,k_2})] \\ & = & z. \end{array}$$

2. Finally, we have the case that f_1 and f_2 are the same function symbol, and there is some value of l with $1 \leq l \leq k_1$, such that $J[T_{1,l}]$ does not equal $J[T_{2,l}]$. Here, we have:

$$J'[f_1(T_{2,1}, \ldots, T_{2,k_2})] = J'(f_1)(J'[T_{2,1}], \ldots, J'[T_{2,k_2}])$$

By property 3, $h(J'[T_{2,i}]) = J[T_{2,i}]$, for all i such that $1 \leq i \leq k_1$. Since $J[T_{1,l}]$ does not equal $J[T_{2,l}]$, the value of the above application of $J'(f_1)$ is:

$$\begin{array}{rcl} J'(f_1)(J'[T_{2,1}], \ldots, J'[T_{2,k_2}]) & = & J(f_1)(h(J'[T_{2,1}]), \ldots, h(J'[T_{2,k_2}])) \\ & = & J(f_1)(J[T_{2,1}], \ldots, J[T_{2,k_2}]) \\ & = & J[f_1(T_{2,1}, \ldots, T_{2,k_2})] \\ & = & z \end{array}$$

□

Lemma 2 *For any interpretation I and p-formula F, there is a maximally diverse interpretation I^* for F such that $I^*[F] \Rightarrow I[F]$.*

Proof: Starting with interpretation I_0 equal to I, we define a sequence of interpretations $I_0, I_1, \ldots$ by repeatedly applying the construction of Lemma 1. That is, we derive each interpretation I_{i+1} from its predecessor I_i by letting $J = I_i$ and letting $I_{i+1} = J'$. Interpretation I_{i+1} is a proper refinement of its predecessor I_i such that $I_{i+1}[F] \Rightarrow I_i[F]$. At some step n, we must reach a maximally diverse interpretation I_n, because our set $\mathcal{T}(F)$ is finite and therefore can only be properly refined a finite number of times. We then let I^* be I_n. We can see that $I^*[F] = I_n[F] \Rightarrow \cdots \Rightarrow I_0[F] = I[F]$, and hence $I^*[F] \Rightarrow I[F]$. □

The completion of the proof of Theorem 1 follows directly from Lemma 2. That is, if we start with any interpretation I for p-formula F, we can construct a maximally diverse interpretation I^* such that $I^*[F] \Rightarrow I[F]$. Assuming F is true under all maximally diverse interpretations, $I^*[F]$ must hold, and since $I^*[F] \Rightarrow I[F]$, $I[F]$ must hold as well.

3.3 Exploiting Positive Equality in a Decision Procedure

A decision procedure for PEUF must determine whether a given p-formula is universally valid. The procedure can significantly reduce the range of possible interpretations it must consider by exploiting the maximal diversity property. Theorem 1 shows that we can consider only interpretations in which the values produced by the application of any p-function symbol differ from those produced by the applications of any other p-function or g-function symbol. We can therefore consider the different p-function symbols to yield values over domains disjoint with one another and with the domain of g-function values. In addition, we can consider each application of a p-function symbol to yield a distinct value, except when its arguments match those of some other application.

4 Eliminating Function Applications

Most work on transforming EUF into propositional logic has used the method described by Ackermann to eliminate applications of functions of nonzero order [Ack54]. In this scheme, each function application term is replaced by a new domain variable and constraints are added to the formula expressing functional consistency. Our approach also introduces new domain variables, but it replaces each function application term with a nested *ITE* structure that directly captures the effects of functional consistency. As we will show, our approach can readily exploit the maximal diversity property, while Ackermann's cannot.

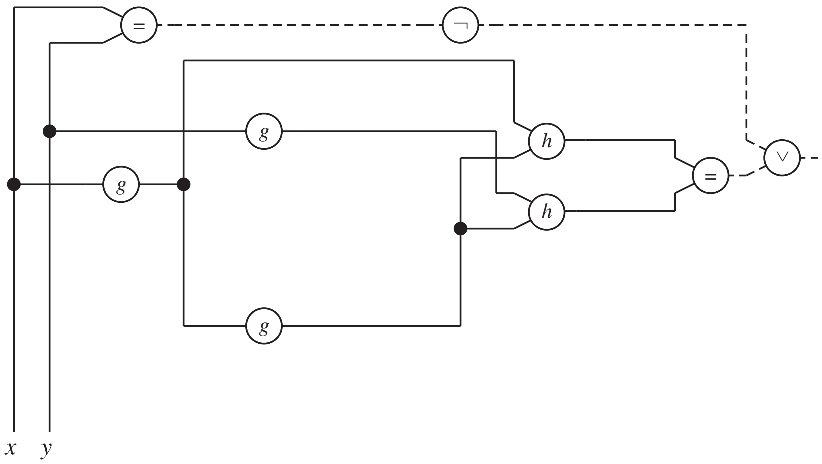
4.1 Function Application Elimination Example

We demonstrate our technique for replacing function applications by domain variables using p-formula F_{eg} (Equation 1) as an example, as illustrated in Figure 4. First consider the three applications of function symbol g: $g(x)$, $g(y)$, and $g(g(x))$, which we identify as terms T_1, T_2, and T_3, respectively. Let vg_1, vg_2, and vg_3 be new domain variables. We generate new terms U_1, U_2, and U_3 as follows:

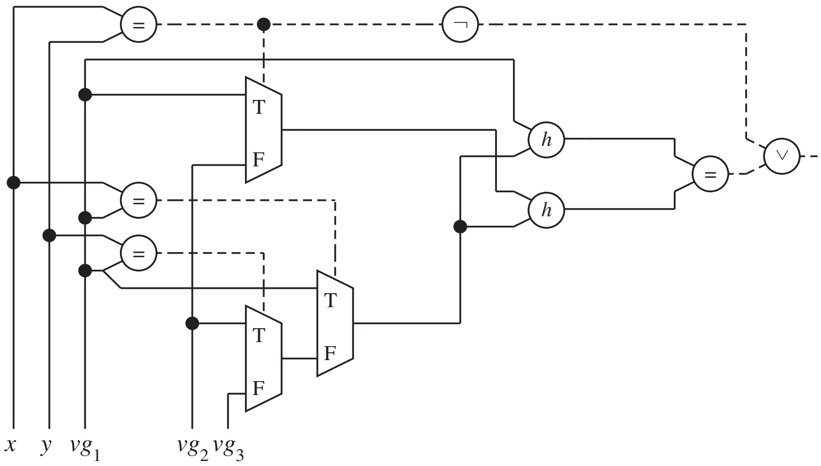
$$\begin{aligned} U_1 &\doteq vg_1 \\ U_2 &\doteq \mathit{ITE}(y = x,\, vg_1,\, vg_2) \\ U_3 &\doteq \mathit{ITE}(vg_1 = x,\, vg_1, \mathit{ITE}(vg_1 = y,\, vg_2,\, vg_3)) \end{aligned} \tag{2}$$

Observe that we use variable vg_1, the translation of $g(x)$, to represent the argument to the outer application of function symbol g in the term $g(g(x))$. In general, we must always process nested applications of a given function symbol working from the innermost to the outermost. Given terms U_1, U_2, and U_3, we eliminate the function applications by replacing each instance of T_i in

Initial formula:



After removing applications of function symbol g:



After removing applications of function symbol h:

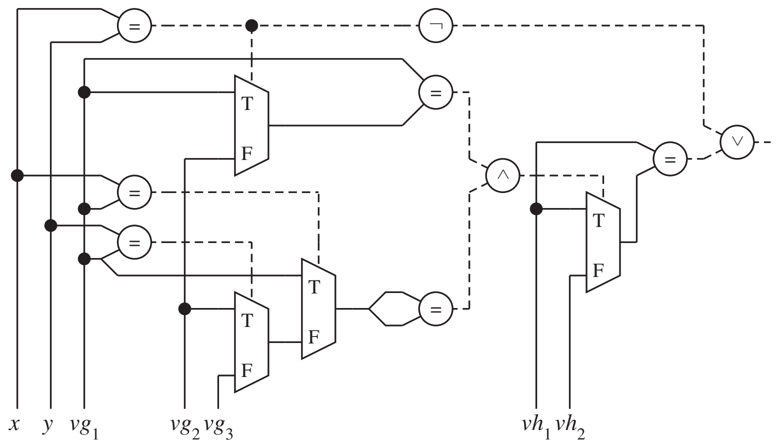


Figure 4: Removing Function Applications from F_{eg}.

$\approx_I$	$I^*[U_1]$	$I^*[U_2]$	$I^*[U_3]$
$\{x\}, \{y\}, \{g(x)\}$	1	2	3
$\{x, y\}, \{g(x)\}$	1	1	3
$\{x\}, \{y, g(x)\}$	1	2	2
$\{x, g(x)\}, \{y\}$	1	2	1
$\{x, y, g(x)\}$	1	1	1

Table 3: Possible valuations of terms in Equation 2 when each variable vg_i is assigned value i.

the formula by U_i for $1 \leq i \leq 3$, as shown in the middle part of Figure 4. We use multiplexors in our schematic diagrams to represent *ITE* operations.

Observe that as we consider interpretations with different values for variables vg_1, vg_2, and vg_3 in Equation 2, we implicitly cover all values that an interpretation of function symbol g in formula F_{eg} may yield for the three arguments. The nested *ITE* structure shown in Equation 2 enforces functional consistency. For example, Table 3 shows the possible valuations of the three terms of Equation 2 for an interpretation I^* assigning values 1, 2, and 3 to domain variables vg_1, vg_2, and vg_3, respectively. For each possible partitioning by I^* of arguments x, y, and $g(x)$ into equivalence classes, we get matching valuations precisely for equivalent arguments.

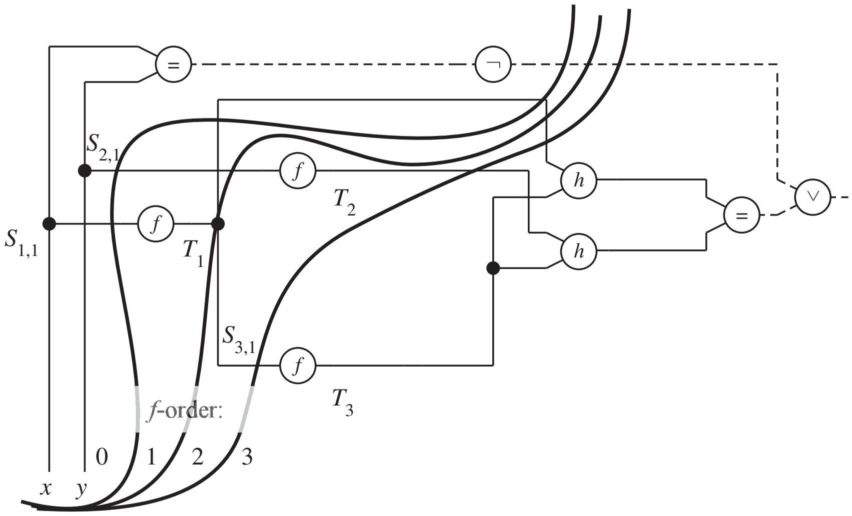
We remove the two applications of function symbol h by a similar process. That is, we introduce two new domain variables vh_1 and vh_2. We replace the first application of h by vh_1 and the second by an *ITE* term that compares the arguments of the two function applications, yielding vh_1 if they are equal and vh_2 if they are not. The final form is illustrated in the bottom part of Figure 4. The translation of predicate applications is similar, introducing a new propositional variable for each application. After removing all applications of function and predicate symbols of nonzero order, we are left with a formula F^*_{eg} containing only domain and propositional variables.

4.2 Algorithm for Eliminating Function and Predicate Applications

The general translation procedure follows the form shown for our example. It iterates through the function and predicate symbols of nonzero order. On each iteration it eliminates all occurrences of a given symbol. At the end we are left with a formula containing only domain and propositional variables.

The following is a detailed description of the process required to eliminate all instances of a single function symbol f having order $k > 0$ from a formula G. We use the variant of formula F_{eg} shown schematically at the top of Figure 5. In this variant, we have replaced function symbol g with f. In the sequel, if E is an expression and T and U are terms, we will write $E[T \leftarrow U]$ for the result of substituting U for each instance of T in E. Let $T_1, \ldots, T_n$ denote the syntactically distinct terms occurring in formula G having the application of f as the top level operation. We

Initial formula showing f-order contours:



After removing applications of function symbol f:

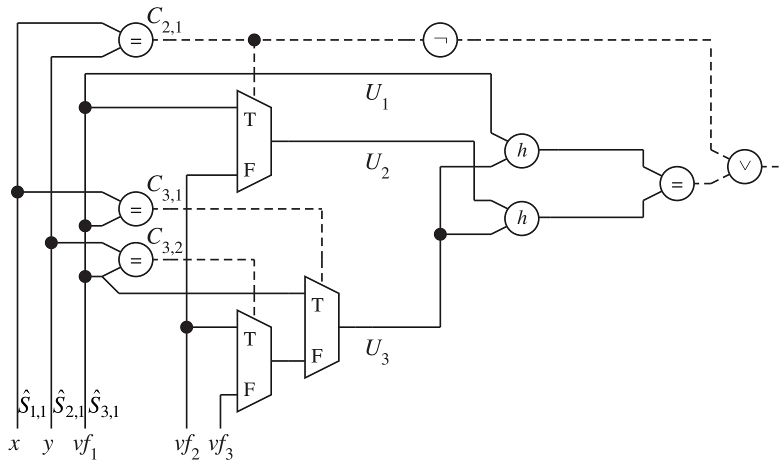


Figure 5: Illustration of Function Application Removal

refer to these as "f-application" terms. Let the arguments to f in f-application term T_i be the terms $S_{i,1}, \ldots, S_{i,k}$, so that T_i has the form $f(S_{i,1}, \ldots, S_{i,k})$. Assume the terms $T_1, \ldots, T_n$ are ordered such that if T_i occurs as a subexpression of T_j then $i < j$. In our example the f-application terms are: $T_1 \doteq f(x)$, $T_2 \doteq f(y)$ and $T_3 \doteq f(f(x))$. These terms have arguments: $S_{1,1} \doteq x$, $S_{2,1} \doteq y$, and $S_{3,1} \doteq f(x)$.

The translation processes the f-application terms in order, such that on step i it replaces all occurrences of the i^{th} application of function symbol f by a nested *ITE* term. Let $vf_1, \ldots, vf_n$ be a new set of domain variables not occurring in F. We use these to encode the possible values returned by the f-application terms.

For any subexpression E in G define its integer-valued f-order, denoted $o_f(E)$, as the highest index i of an f-application term T_i occurring in E. If no f-application terms occur in E, its f-order is defined to be 0. By our ordering of the f-application terms, any argument $S_{i,l}$ to f-application term T_i must have $o_f(S_{i,l}) < o_f(T_i)$, and therefore $o_f(T_i) = i$. For example, the contour lines shown in Figure 5 partition the operators according to their f-order values.

The transformations performed in replacing applications of function symbol f can be expressed by defining the following recurrence for any subexpression E of G:

$$\begin{array}{rcll} E^{(0)} & \doteq & E & \\ E^{(i)} & \doteq & E^{(i-1)}[T_i^{(i-1)} \leftarrow U_i], & 1 \leq i \leq n \\ \hat{E} & \doteq & E^{(m)}, & \text{where } m = o_f(E) \end{array} \tag{3}$$

In this equation, term $T_i^{(i-1)}$ is the form of the i^{th} f-application term T_i after all but the topmost application of f have been eliminated. Term U_i is a nested *ITE* structure encoding the possible values returned by T_i while enforcing its consistency with earlier applications. U_i does not contain any applications of function symbol f. For a subexpression E with $o_f(E) = m$, its form $E^{(m)}$ will contain no applications of function symbol f. We denote this form as $\hat{E}$. Observe that for any $i > o_f(E)$, term $T_i^{(i-1)}$ does not occur in $E^{(i)}$, and hence $E^{(i)} = \hat{E}$ for all $i \geq o_f(E)$. Observe also that for f-application term T_i, we have $\hat{T_i} = T_i^{(i)} = U_i$.

U_i is defined in terms of a recursively-defined term $V_{i,j}$ as follows:

$$\begin{array}{rcll} V_{i,i} & \doteq & vf_i, & 1 \leq i \leq n \\ V_{i,j} & \doteq & \textit{ITE}(C_{i,j}, vf_j, V_{i,j+1}), & 1 \leq j < i \leq n \\ U_i & \doteq & V_{i,1}, & 1 \leq i \leq n \end{array} \tag{4}$$

where for each $j < i$, formula $C_{i,j}$ is true iff the (transformed) arguments to the top-level application of f in the terms T_i and T_j have the same values:

$$C_{i,j} \quad \doteq \quad \bigwedge_{1 \leq l \leq k} \hat{S}_{i,l} = \hat{S}_{j,l} \tag{5}$$

Observe that the recurrence of Equation 4 is well-defined, since for all argument terms of the form $S_{j,l}$ for $1 \leq j \leq i$ and $1 \leq l \leq k$, we have $o_f(S_{j,l}) < i$, and hence terms of the form $\hat{S}_{j,l}$ and $\hat{S}_{i,l}$, as well as term $V_{i,j+1}$ are available when we define $V_{i,j}$.

The lower part of Figure 5 shows the result of removing the three applications of f from our example formula. First, we have $U_1 \doteq vf_1$, giving translated function arguments: $\hat{S}_{1,1} \doteq x$, $\hat{S}_{2,1} \doteq y$, and $\hat{S}_{3,1} \doteq vf_1$. The comparison formulas are then: $C_{2,1} \doteq (y = x)$, $C_{3,1} \doteq (vf_1 = x)$, and $C_{3,2} \doteq (vf_1 = y)$. From these we get translated terms:

$$\begin{array}{rcl} U_2 & \doteq & \mathit{ITE}(y = x, vf_1, vf_2) \\ U_3 & \doteq & \mathit{ITE}(vf_1 = x, vf_1, \mathit{ITE}(vf_1 = y, vf_2, vf_3)) \end{array}$$

We can see that formula $\hat{G} \doteq G^{(n)}$ will no longer contain any applications of function symbol f. We will show that $\hat{G}$ is universally valid if and only if G is.

In the following correctness proofs, we will use a fundamental principle relating syntactic substitution and expression evaluation:

Proposition 1 *For any expression E, pair of terms T, U, and interpretation I of all of the symbols in E, T, and U, if $I[T] = I[U]$ then $I[E[T \leftarrow U]] = I[E]$.*

We will also use the following characterization of Equation 4. For value i such that $1 \leq i \leq n$ and for interpretation I of the symbols in U_i, we define the *least matching value* of i under interpretation I, denoted $lm_I(i)$, as the minimum value j in the range $1 \leq j \leq i$ such that $I[\hat{S}_{j,l}] = I[\hat{S}_{i,l}]$ for all l in the range $1 \leq l \leq k$. Observe that this value is well defined, since i forms a feasible value for j in any case.

Lemma 3 *For any interpretation I, $I[U_i] = I(vf_j)$, where $j = lm_I(i)$.*

Proof: For value m in the range $1 \leq m \leq i$ define $lm_I(m, i)$ as the minimum value of j in the range $m \leq j \leq i$ such that $I[\hat{S}_{j,l}] = I[\hat{S}_{i,l}]$ for all l in the range $1 \leq l \leq k$. By this definition $lm_I(i) = lm_I(1, i)$. Observe also that if $j = lm_I(m, i)$ then $I[C_{i,j}] = \mathtt{true}$. In addition, for any value m' in the range $m \leq m' \leq i$, if $lm_I(m, i) \geq m'$, then $lm_I(m, i) = lm_I(m', i)$.

We prove by induction on m that $I[V_{i,m}] = I(vf_j)$, where $j = lm_I(m, i)$. The base case of $m = i$ is trivial, since $lm_I(i, i) = i$, and $V_{i,i} = vf_i$.

Assuming the property holds for $m + 1$, we consider two possibilities. First, if $lm_I(m, i) = m$, we have $I[C_{i,m}] = \mathtt{true}$, and hence the top-level *ITE* operation in $V_{i,m}$ (Equation 4) will select its first term argument vf_m, giving $I[V_{i,m}] = I(vf_m)$. On the other hand, if $lm_I(m, i) > m$, we must have $I[C_{i,m}] = \mathtt{false}$, and hence the top-level *ITE* operation in $V_{i,m}$ will select its second term argument $V_{i,m+1}$, giving $I[V_{i,m}] = I[V_{i,m+1}]$, which by the inductive hypothesis equals $I(vf_j)$ for $j = lm_I(m + 1, i)$. Since $lm_I(m, i) \geq m + 1$, we must also have $lm_I(m, i) = lm_I(m + 1, i)$, and hence $I[V_{i,m}] = I(vf_j)$, where $j = lm_I(m, i)$.

Since U_i is defined as $V_{i,1}$, our induction argument proves that $I[U_i] = I(vf_j)$ for $j = lm_I(1, i) = lm_I(i)$. $\Box$

Lemma 4 *Any interpretation J of the symbols in G can be extended to an interpretation $\hat{J}$ of the symbols in both G and $\hat{G}$ such that for every subexpression E of G, $\hat{J}[\hat{E}] = \hat{J}[E] = J[E]$.*

Proof: We provide a somewhat more general construction of $\hat{J}$ than is required for the proof of this lemma in anticipation of using this construction in the proof of Lemma 6. Given J defined over domain $\mathcal{D}$, we define $\hat{J}$ over a domain $\hat{\mathcal{D}}$ such that $\hat{\mathcal{D}} \supseteq \mathcal{D}$.

We define $\hat{J}$ for the function and predicate symbols occurring in G based on their definitions in J. For any function symbol f in G having $ord(f) = k$, and any argument values $x_1, \ldots, x_k \in \mathcal{D}$, we define $\hat{J}(f)(x_1, \ldots, x_k) \doteq J(f)(x_1, \ldots, x_k)$. For argument values $x_1, \ldots, x_k \in \hat{\mathcal{D}}$ such that for some i, $x_i \notin \mathcal{D}$, we let $\hat{J}(f)(x_1, \ldots, x_k)$ be an arbitrary domain value. Similarly, for predicate symbol p, we define $\hat{J}(p)$ to yield the same value as $J(p)$ for arguments in $\mathcal{D}$ and to yield an arbitrary truth value when at least one argument is not in $\mathcal{D}$.

One can readily see that $\hat{J}[E] = J[E]$ for every subexpression E of G. This takes care of the second equality in the statement of the lemma, and hence we can concentrate on the relation between $\hat{J}[\hat{E}]$ and $\hat{J}[E]$ for the remainder of the proof.

Recall that $vf_1, \ldots, vf_n$ are the domain variables introduced when generating the nested *ITE* terms $U_1, \ldots U_n$. Our strategy is to define interpretations of these variables such that each U_i mimics the behavior of the original f-application term T_i in G.

We consider two cases. For the case where $lm_{\hat{J}}(i) = i$, we define $\hat{J}(vf_i) = \hat{J}[T_i]$, i.e., the value of the i^{th} f-application term in G under J. Otherwise, we let $\hat{J}(vf_i)$ be an arbitrary domain value—we will show that its value does not affect the valuation of any expression $\hat{E}$ in $\hat{G}$ having a counterpart E in G.

We argue by induction on i that $\hat{J}[E^{(i)}] = \hat{J}[E]$ for any subexpression E of G. For the case where $o_f(E) \leq i$, this hypothesis implies that $\hat{J}[\hat{E}] = \hat{J}[E]$. The base case of $i = 0$ is trivial, since $E^{(0)}$ is defined to be E.

Suppose that for every j in the range $1 \leq j < i$ and every subexpression D of G, we have $\hat{J}[D^{(j)}] = \hat{J}[D]$, and consequently that $\hat{J}[\hat{D}] = \hat{J}[D]$ for the case where $o_f(D) < i$. We must show that for every subexpression E of G, we have $\hat{J}[E^{(i)}] = \hat{J}[E]$.

We first focus our attention on term T_i in G and its counterpart U_i in $\hat{G}$, showing that $\hat{J}[U_i] = \hat{J}[T_i]$. The f-application terms for all j such that $j < i$ have $o_f(T_j) = j < i$, and hence we can assume that $\hat{J}[U_j] = \hat{J}[T_j]$ for these values of j. Furthermore, any argument $S_{j,l}$ to an f-application term for $j \leq i$ and $1 \leq l \leq k$ has $o_f(S_{j,l}) < j \leq i$, and hence we can assume $\hat{J}[\hat{S}_{j,l}] = \hat{J}[S_{j,l}]$.

We consider two cases: $lm_{\hat{J}}(i) = i$, and $lm_{\hat{J}}(i) < i$. In the former case, we have by Lemma 3 that $\hat{J}[U_i] = \hat{J}(vf_i)$. Our definition of $\hat{J}(vf_i)$ gives $\hat{J}[U_i] = \hat{J}(vf_i) = \hat{J}[T_i]$. Otherwise, suppose that $lm_{\hat{J}}(i) = j < i$. Lemma 3 shows that $\hat{J}[U_i] = \hat{J}(vf_j)$. We can see that $lm_{\hat{J}}(j) = j$, and hence $\hat{J}(vf_j)$ is defined to be $\hat{J}[T_j]$. By the definition of lm we have $\hat{J}[\hat{S}_{j,l}] = \hat{J}[\hat{S}_{i,l}]$ for $1 \leq l \leq k$. By the induction hypothesis we have $\hat{J}[\hat{S}_{j,l}] = \hat{J}[S_{j,l}]$, since $o_f(S_{j,l}) < i$, and similarly that $\hat{J}[\hat{S}_{i,l}] = \hat{J}[S_{i,l}]$. By transitivity we have $\hat{J}[S_{j,l}] = \hat{J}[S_{i,l}]$ for all l such that $1 \leq l \leq k$, i.e., the arguments to f-application terms T_j and T_i have equal valuations under J. Function consistency requires that $\hat{J}[T_j] = \hat{J}[T_i]$. From this we can conclude that $\hat{J}[U_i] = \hat{J}[U_j] = \hat{J}[T_j] = \hat{J}[T_i]$. Combining these cases gives $\hat{J}[U_i] = \hat{J}[T_i]$.

For any subexpression E its form $E^{(i)}$ differs from $E^{(i-1)}$ only in that all instances of term $T_i^{(i-1)}$ have been replaced by U_i. We have just argued that $\hat{J}[U_i] = \hat{J}[T_i]$, and by the induction hypothesis we have that $\hat{J}[T_i^{(i-1)}] = \hat{J}[T_i]$, giving by transitivity that $\hat{J}[T_i^{(i-1)}] = \hat{J}[U_i]$. Proposition 1 implies that $\hat{J}[E^{(i)}] = \hat{J}[E^{(i-1)}]$, and our induction hypothesis gives $\hat{J}[E^{(i-1)}] = \hat{J}[E]$. By transitivity we have $\hat{J}[E^{(i)}] = \hat{J}[E]$.

To complete the proof, we observe that our induction argument implies that for any subexpression E of G, $\hat{J}[E^{(m)}] = \hat{J}[E]$, including for the case where $m = o_f(E)$, giving $\hat{J}[\hat{E}] = \hat{J}[E^{(m)}] = \hat{J}[E]$. □

Lemma 5 *Any interpretation $\hat{J}$ of the symbols in $\hat{G}$ can be extended to an interpretation J of the symbols in both $\hat{G}$ and G such that for every subexpression E of G, $J[E] = J[\hat{E}] = \hat{J}[\hat{E}]$.*

Proof: We define J to be identical to $\hat{J}$ for any symbol occurring in $\hat{G}$. This implies that $J[\hat{E}] = \hat{J}[\hat{E}]$ for every subexpression E of G. This takes care of the second equality in the statement of the lemma, and hence we can concentrate on the relation between $J[E]$ and $J[\hat{E}]$ for the remainder of the proof.

For function symbol f, we define $J(f)(x_1, \ldots, x_k)$ for domain elements $x_1, \ldots, x_k$ as follows. Suppose there is some value j such that $x_l = J[\hat{S}_{j,l}]$ for all l such that $1 \leq l \leq k$, and such that $j = lm_{\hat{J}}(j)$. Then we define $J(f)(x_1, \ldots, x_k)$ to be $J(vf_j)$. If no such value of j exists, we let $J(f)(x_1, \ldots, x_k)$ be some arbitrary domain value.

We argue by induction on i that $J[E] = J[E^{(i)}]$ for any subexpression E of G. For the case where $o_f(E) \leq i$, this hypothesis implies that $J[E] = J[\hat{E}]$. The base case of $i = 0$ is trivial, since $E^{(0)}$ is defined to be E.

Suppose that for every j in the range $1 \leq j < i$ and every subexpression D of G, we have $J[D] = J[D^{(i)}]$, and consequently that $J[D] = J[\hat{D}]$ for the case where $o_f(D) < i$. We must show that for every subexpression E of G, we have $J[E] = J[E^{(i)}]$.

We focus initially on term T_i in G and its counterpart U_i in $\hat{G}$, showing that $J[T_i] = J[U_i]$. Any f-application term T_j for $j < i$ has $o_f(T_j) = j < i$, and hence we can assume that $J[T_j] = J[\hat{T}_j]$. Furthermore, any argument $S_{j,l}$ to an f-application term for $j \leq i$ and $1 \leq l \leq k$ has $o_f(S_{j,l}) < j \leq i$, and hence we can assume that $J[S_{j,l}] = J[\hat{S}_{j,l}]$.

We consider two cases: $lm_{\hat{J}}(i) = i$, and $lm_{\hat{J}}(i) < i$. In the former case, we have by Lemma 3 that $J[U_i] = J(vf_i)$. In addition, $J(f)$ is defined such that $J[T_i] = J(f)(J[S_{i,1}], \ldots, J[S_{i,k}]) = J(f)(J[\hat{S}_{i,1}], \ldots, J[\hat{S}_{i,k}]) = J(vf_i)$, giving $J[T_i] = J(vf_i) = J[U_i]$. Otherwise, suppose that $lm_J(i) = j < i$. Lemma 3 shows that $J[U_i] = J(vf_j)$. We can see that $lm_{\hat{J}}(j) = j$, and hence

$J(f)$ is defined such that $J(f)(J[\hat{S}_{j,1}], \ldots, J[\hat{S}_{j,k}]) = J(vf_j)$. For any l such that $1 \leq l \leq k$, we also have by the definition of lm that $J[\hat{S}_{j,l}] = J[\hat{S}_{i,l}]$. By the induction hypothesis we have $J[S_{j,l}] = J[\hat{S}_{j,l}]$, since $o_f(S_{j,l}) < i$, and similarly that $J[S_{i,l}] = J[\hat{S}_{i,l}]$. By transitivity we have $J[S_{j,l}] = J[S_{i,l}]$, i.e., the arguments to f-application terms T_j and T_i have equal valuations under J. Functional consistency requires that $J[T_j] = J[T_i]$. Putting this together gives $J[T_i] = J[T_j] = J(f)(J[S_{j,1}], \ldots, J[S_{j,k}]) = J(f)(J[\hat{S}_{j,1}], \ldots, J[\hat{S}_{j,k}]) = J(vf_j) = J[U_i]$.

For any subexpression E its form $E^{(i)}$ differs from $E^{(i-1)}$ only in that all instances of term $T_i^{(i-1)}$ have been replaced by U_i. We have just argued that $J[T_i] = J[U_i]$, and by the induction hypothesis we have that $J[T_i] = J[T_i^{(i-1)}]$, giving by transitivity that $J[T_i^{(i-1)}] = J[U_i]$. Proposition 1 implies that $J[E^{(i-1)}] = J[E^{(i)}]$, and our induction hypothesis gives $J[E] = J[E^{(i-1)}]$. By transitivity we have $J[E] = J[E^{(i)}]$.

To complete the proof, we observe that our induction argument implies that for any subexpression E of G, $J[E] = J[E^{(m)}]$, including for the case where $m = o_f(E)$, giving $J[E] = J[E^{(m)}] = J[\hat{E}]$. $\Box$

An application of a predicate symbol having nonzero order can be removed by a similar process, using newly generated propositional variables to encode the possible values returned by the predicate applications. By an argument similar to that made in Lemma 4, we can extend an interpretation to include interpretations of the propositional variables such that the original and the transformed formulas have identical valuations. Conversely, by an argument similar to that made in Lemma 5, we can extend an interpretation to include an interpretation of the original predicate symbol such that the original and the transformed formulas have identical valuations.

Suppose formula F contains applications m different function and predicate symbols of nonzero order. Starting with $F_0 \doteq F$, we can generate a sequence of formulas $F_0, F_1, \ldots, F_m$. Each formula F_i is generated from its predecessor F_{i-1} by letting $G = F_i$ and $F_{i+1} = \hat{G}$ in our technique to eliminate all instances of the i^{th} function or predicate symbol. Let $F^* \doteq F_m$ denote the formula that will result once we have eliminated all applications of function and predicate symbols having nonzero order.

Theorem 2 *For EUF formula F, the transformation process described above yields a formula F^* such that F is universally valid if and only if F^* is universally valid.*

Proof: This theorem follows by simply inducting on the number of function and predicate symbols in F having nonzero order. That is, for any interpretation I of the function and predicate symbols of F, we construct a sequence of interpretations $I = I_0, I_1, \ldots, I_m$. Each interpretation I_i is generated by extending its predecessor I_{i-1} by letting $J = I_{i-1}$ and $I_i = \hat{J}$ in Lemma 4 or a similar one for predicate applications. The effect is to include in I_i interpretations of the domain or propositional variables introduced when eliminating the i^{th} function or predicate symbol. We then define interpretation I^* to be identical to I_m for every variable appearing in F^*. By induction, we have $I^*[F^*] = I[F]$. If F^* is universally valid, we have $I[F] = I^*[F^*] = \texttt{true}$. Since this construction can be performed for any interpretation I, F must also be universally valid.

Conversely, starting with an interpretation I^* of the domain and propositional variables of F^*, we can define a sequence of interpretations $I^* = I_m, I_{m-1}, \ldots, I_0$, using the construction in the proof of Lemma 5 (or a similar one for predicate applications) to generate an interpretation of each function or predicate symbol in F. We then define interpretation I to be identical to I_0 for every function or predicate symbol appearing in F. By induction, we have $I[F] = I^*[F^*]$. If F is universally valid, we have $I^*[F^*] = I[F] = \texttt{true}$. Since this construction can be performed for any interpretation I^*, F^* must also be universally valid. $\Box$

4.3 Assigning Distinct Values to Variables Representing P-Function Applications

We can exploit the maximal diversity property by considering only interpretations that assign distinct values to the domain variables generated when replacing p-function applications by nested *ITE* terms.

For example, by using an interpretation I^* that assigns distinct values 1, 2, and 3 to variables vf_1, vf_2, and vf_3 in Equation 2, we generate distinct values for the terms U_1, U_2, and U_3, except when there are matches between the arguments x_1, x_2, and x_3. On the other hand, our encoding still considers the possibility that the arguments to the different applications of f may match under some interpretations, in which case the function results should match as well.

To show this formally, consider the effect of replacing all instances of a function symbol f in a formula G by nested *ITE* terms, as described earlier, yielding a formula $\hat{G}$ with new domain variables $vf_1, \ldots, vf_n$. We first show that when we generate these variables while eliminating p-function applications, we can assume they have a diverse interpretation.

Lemma 6 *Let Σ be a subset of the symbols in G, and let $\hat{G}$ be the result of eliminating function symbol f from G by introducing new domain variables $vf_1, \ldots, vf_n$. If $f \in \Sigma$, then for any interpretation J that is diverse for G with respect to Σ, there is an interpretation $\hat{J}$ that is diverse for $\hat{G}$ with respect to $\Sigma - \{f\} \cup \{vf_1, \ldots, vf_n\}$ such that $\hat{J}[\hat{G}] = J[G]$.*

Proof: Given interpretation J defined over domain $\mathcal{D}$, we define interpretation $\hat{J}$ over a domain $\hat{\mathcal{D}} \doteq \mathcal{D} \cup \{z_1, \ldots, z_n\}$. Each z_i is a unique value, i.e., $z_i \neq z_j$ for any $i \neq j$, and $z_i \notin \mathcal{D}$.

The proof of this lemma is based on a refinement of the proof of Lemma 4. Whereas the construction in the earlier proof assigned arbitrary values to the new domain variables in some cases, we select an assignment that is diverse in these variables. As in the construction in the proof of Lemma 4, we define $\hat{J}$ for any function or predicate symbol in G to be identical to that of J when the arguments are all elements of $\mathcal{D}$. When some argument is not in $\mathcal{D}$, we let the function (respectively, predicate) application yield an arbitrary domain (resp., truth) value.

For domain variable vf_i introduced when generating term U_i, we consider two cases. For the case where $lm_{\hat{J}}(i) = i$, we define $\hat{J}(vf_i) = \hat{J}[T_i]$, i.e., the value of the i^{th} f-application term in G under J. For the case where $lm_{\hat{J}}(i) < i$, we define $\hat{J}(vf_i) = z_i$. We saw in the proof of Lemma 4 that we could assign arbitrary values in this latter case and still have $\hat{J}[\hat{G}] = J[G]$. In fact, for every subexpression E of G, we have that its counterpart $\hat{E}$ in $\hat{G}$ satisfies $\hat{J}[\hat{E}] = J[E]$.

We must show that $\hat{J}$ is diverse for $\hat{G}$ with respect to $\Sigma - \{f\} \cup \{vf_1, \ldots, vf_n\}$. We first observe that $\hat{J}$ is identical to J for all function application terms in G, and hence $\hat{J}$ must be diverse with respect to Σ for G. We also observe that $\hat{J}$ assigns to each variable vf_i either a unique value z_i or the value yielded by f-application term T_i in G under $\hat{J}$.

Suppose there were distinct variables vf_i and vf_j such that $\hat{J}[vf_i] = \hat{J}[vf_j]$. This could only occur for the case that $\hat{J}(vf_i) = \hat{J}[T_i] = \hat{J}[T_j] = \hat{J}(vf_j)$, but this would imply that $lm_{\hat{J}}(i) = lm_{\hat{J}}(j)$. We cannot have both $lm_{\hat{J}}(i) = i$ and $lm_{\hat{J}}(j) = j$, and hence either vf_i or vf_j would have been assigned unique value z_i or z_j, respectively. Thus, we can conclude that $\hat{J}[vf_i] \neq \hat{J}[vf_j]$ for distinct variables vf_i and vf_j.

In addition, we must show that interpretation $\hat{J}$ does not create any matches between a new variable vf_i and a function application term T in G that does not have f as the topmost function symbol. Since $\hat{J}$ is diverse with respect to Σ for G and $f \in \Sigma$, any function application term T in G that does not have function symbol f as its topmost symbol must have $\hat{J}[T] \neq \hat{J}[T_i]$ for all $1 \leq i \leq n$. In addition, we have $\hat{J}[T] \neq z_i$ for all $1 \leq i \leq n$. Hence, we must have $\hat{J}[T] \neq J(vf_i)$. □

We must also show that the variables introduced when eliminating g-function applications do not adversely affect the diversity of the other symbols.

Lemma 7 *Let Σ be a subset of the symbols in G, and let $\hat{G}$ be the result of eliminating function symbol f from G by introducing new domain variables $vf_1, \ldots, vf_n$. If $f \notin \Sigma$, then for any interpretation J that is diverse for G with respect to Σ, there is an interpretation $\hat{J}$ that is diverse for $\hat{G}$ with respect to Σ such that $\hat{J}[\hat{G}] = J[G]$.*

Proof: The proof of this lemma is based on a refinement of the proof of Lemma 4. Whereas the construction in the earlier proof assigned arbitrary values to some of the new domain variables, we select an assignment such that we do not inadvertently violate the diversity of the other function symbols.

We define $\hat{J}$ to be identical to J for any symbol occurring in G. For each domain variable vf_i introduced when generating term U_i, we define $\hat{J}(vf_i) = \hat{J}[T_i]$. This differs from the interpretation defined in the proof of Lemma 4 only in that give fixed interpretations of domain variables that could otherwise be arbitrary, and hence we have have $\hat{J}[\hat{G}] = J[G]$. In fact, for every subexpression E of G, we have that its counterpart $\hat{E}$ in $\hat{G}$ satisfies $\hat{J}[\hat{E}] = J[E]$.

We must show that $\hat{J}$ is diverse for $\hat{G}$ with respect to Σ. We first observe that $\hat{J}$ is identical to J for all function application terms in G, and hence $\hat{J}$ must be diverse for G with respect to

Σ. We also observe that $\hat{J}$ assigns to each variable vf_i the value of f-application term T_i. For term T having the application of function symbol $g \in \Sigma$ as the topmost operation, we must have $\hat{J}[\hat{T}] = \hat{J}[T] \neq \hat{J}[T_i] = J[vf_i]$. Hence, we are assured that the values assigned to the new variables under $\hat{J}$ do not violate the diversity of the interpretations of the symbols in Σ. $\Box$

Suppose we apply the transformation process of Theorem 2 to a p-formula F to generate a formula F^*, and that in this process, we introduce a set of new domain variables V to replace the applications of the p-function symbols. Let $\Sigma_p^*(F)$ be the union of the set of domain variables in $\Sigma_p(F)$ and V. That is, $\Sigma_p^*(F)$ consists of those domain variables in the original formula F that were p-function symbols as well as the domain variables generated when replacing applications of p-function symbols. Let $\Sigma_g^*(F)$ be the domain variables in F^* that are not in $\Sigma_p^*(F)$. These variables were either g-function symbols in F or were generated when replacing g-function applications.

We observe that we can generate all maximally diverse interpretations of F by considering only interpretations of the variables in F^* that assign distinct values to the variables in $\Sigma_p^*(F)$:

Theorem 3 *PEUF formula F is universally valid if and only if its translation F^* is true for every interpretation I^* that is diverse over $\Sigma_p^*(F)$.*

Proof: By Theorem 2, the universal validity of F implies that of F^*. The theorem follows by inducting on the number of function and predicate symbols in F having nonzero order. For the induction step we use Lemma 6 when eliminating all applications of a p-function symbol, and Lemma 7 when eliminating all applications of a g-function symbol. When eliminating a predicate symbol, we do not introduce any new domain variables. $\Box$

4.3.1 Discussion

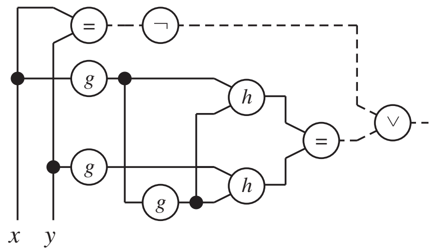
Ackermann also describes a scheme for replacing function application terms by domain variables [Ack54]. His scheme simply replaces each instance of a function application by a newly-generated domain variable and then introduces constraints expressing functional consistency as antecedents to the modified formula. As an illustration, Figure 6 shows the result of applying his method to formula F_{eg} of Equation 1. First, we replace the three applications of function symbol g with new domain variables vg_1, vg_2, and vg_3. To maintain functional consistency we add constraints

$$(x = y \Rightarrow vg_1 = vg_2) \wedge (x = vg_1 \Rightarrow vg_1 = vg_3) \wedge (y = vg_1 \Rightarrow vg_2 = vg_3)$$

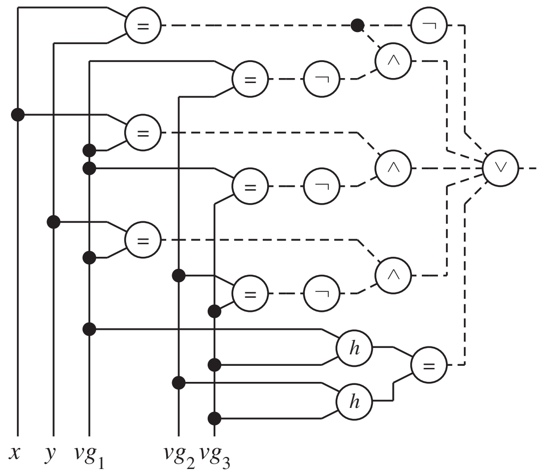
as an antecedent to the modified formula. The result is shown in the middle of Figure 6, using Boolean connectives $\wedge$, $\vee$, and $\neg$ rather than $\Rightarrow$. In this diagram, the three constraints listed above form the middle three arguments of the final disjunction. A similar process is used to replace the applications of function symbol h, adding a fourth constraint $vg_1 = vg_2 \wedge vg_3 = vg_3 \Rightarrow vh_1 = vh_2$. The result is shown at the bottom of Figure 6.

There is no clear way to exploit the maximal diversity with this translated form. For example, if we consider only diverse interpretations of variables vg_1, vg_2, and vg_3, we will fail to consider interpretations of the original formula for which x equals y.

Initial formula:



After removing applications of function symbol g:



After removing applications of function symbol h:

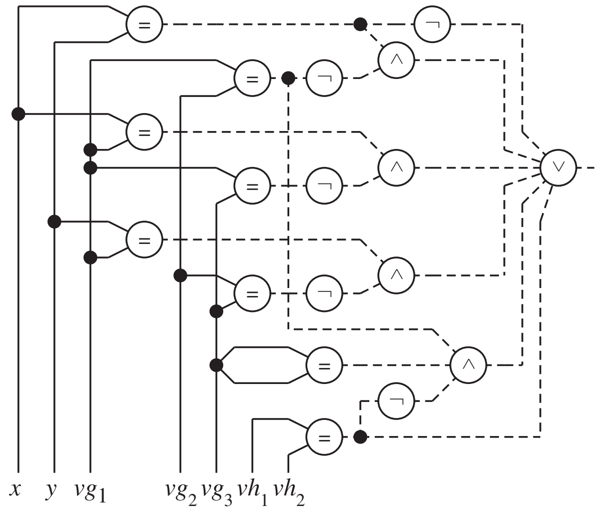


Figure 6: Ackermann's Method for Replacing Function Applications in F_{eg}.

4.4 Using Fixed Interpretations of the Variables in $\Sigma_p^*(F)$

We can further simplify the task of determining universal validity by choosing particular domains of sufficient size and assigning fixed interpretations to the variables in $\Sigma_p^*(F)$. The next result follows from Theorem 3.

Corollary 1 *Let $\mathcal{D}_p$ and $\mathcal{D}_g$ be disjoint subsets of domain $\mathcal{D}$ such that $|\mathcal{D}_p| \geq |\Sigma_p^*(F)|$ and $|\mathcal{D}_g| \geq |\Sigma_g^*(F)|$. Let α be any 1–1 mapping $\alpha\colon \Sigma_p^*(F) \rightarrow \mathcal{D}_p$. PEUF formula F is universally valid if and only if its translation F^* is true for every interpretation I^* such that $I^*(v_p) = \alpha(v_p)$ for every variable $v_p \in \Sigma_p^*(F)$, and $I^*(v_g) \in \mathcal{D}_g$ for every variable $v_g \in \Sigma_g^*(F)$.*

Proof: Consider any interpretation J^* of the variables in $\Sigma_p^*(F) \cup \Sigma_g^*(F)$ that is diverse over $\Sigma_p^*(F)$. We show that we can construct an isomorphic interpretation I^* that satisfies the restrictions of the corollary.

Let $\mathcal{D}'_p$ (respectively, $\mathcal{D}'_g$) be the range of J^* considering only variables in $\Sigma_p^*(F)$ (resp., $\Sigma_g^*(F)$). The function $J^*\colon \Sigma_p^*(F) \rightarrow \mathcal{D}'_p$ must be a bijection and hence have an inverse $J^{*-1}\colon \mathcal{D}'_p \rightarrow \Sigma_p^*(F)$. Furthermore, we must have $|\mathcal{D}'_g| \leq |\Sigma_g^*(F)| \leq |\mathcal{D}_g|$. Let σ_p be the 1–1 mapping $\sigma_p\colon \mathcal{D}'_p \rightarrow \mathcal{D}_p$ defined for any z in $\mathcal{D}'_p$, as $\sigma_p(z) = \alpha(J^{*-1}(z))$. Let σ_g be an arbitrary 1–1 mapping $\sigma_g\colon \mathcal{D}'_g \rightarrow \mathcal{D}_g$. We now define I^* such that for any variable v in $\Sigma_p^*(F)$ (respectively, $\Sigma_g^*(F)$) we have $I^*(v)$ equal to $\sigma_p(J^*(v))$ (resp., $\sigma_g(J^*(v))$). Finally, for any propositional variable a, we let $I^*(a)$ equal $J^*(a)$.

For any EUF formula, isomorphic interpretations will always yield identical valuations, giving $I^*[F^*] = J^*[F^*]$. Hence the set of interpretations satisfying the restrictions of the corollary form a sufficient set to prove the universal validity of F^*. $\square$

5 Reductions to Propositional Logic

We present two different methods of translating a PEUF formula into a propositional formula that is tautological if and only if the original formula is universally valid. Both use the function and predicate elimination method described in the previous section so that the translation can be applied to a formula F^* containing only domain and predicate variables. In addition, we assume that a subset of the domain variables $\Sigma_p^*(F)$ has been identified such that we only need to encode interpretations that are diverse over these variables.

5.1 Translation Based on Bit Vector Interpretations

A formula such as F^* containing only domain and propositional variables can readily be translated into one in propositional logic, using the set of bit vectors of some length k greater than or equal to $\log_2 m$ as the domain of interpretation for a formula containing m domain variables [VB98].

Domain variables are represented with vectors of propositional variables. In this formulation, we represent a domain variable as a vector of propositional variables, where truth value `false` encodes bit value 0, and truth value `true` encodes bit value 1. In [VB98] we described an encoding scheme in which the i^{th} domain variable is encoded as a bit vector of the form $\langle 0, \ldots, 0, a_{i,k-1}, \ldots, a_{i,0} \rangle$ where $k = \lceil \log_2 i \rceil$, and each $a_{i,j}$ is a propositional variable. This scheme can be viewed as encoding interpretations of the domain variables over the integers where the i^{th} domain variable ranges over the set $\{0, \ldots, i-1\}$ [PRSS99]. That is, it may equal any of its predecessors, or it may be distinct.

We then recursively translate F^* using vectors of propositional formulas to represent terms. By this means we then reduce F^* to a propositional formula that is tautological if and only if F^*, and consequently the original EUF formula F, is universally valid.

We can exploit positive equality by using fixed bit vectors, rather than vectors of propositional variables when encoding variables in $\Sigma_p^*(F)$. Furthermore, we can construct our bit encodings such that the vectors encoding variables in $\Sigma_g^*(F)$ never match the bit patterns encoding variables in $\Sigma_p^*(F)$. As an illustration, consider formula F_{eg} given by Equation 1 translated into formula F_{eg}^* as diagrammed at the bottom of Figure 4. We need only encode interpretations of the variables x, y, vg_1, vg_2, vg_3, vh_1, and vh_2 that are diverse respect to the last five variables. Therefore, we can assign 3-bit encodings to the seven variables as follows:

x	$\langle 0, 0, 0 \rangle$
y	$\langle 0, 0, a_{1,0} \rangle$
vg_1	$\langle 0, 1, 0 \rangle$
vg_2	$\langle 0, 1, 1 \rangle$
vg_3	$\langle 1, 0, 0 \rangle$
vh_1	$\langle 1, 0, 1 \rangle$
vh_2	$\langle 1, 1, 0 \rangle$

where $a_{1,0}$ is a propositional variable. This encoding uses the same scheme as [VB98] for the variables in $\Sigma_g^*(F)$ but uses fixed bit patterns for the variables in $\Sigma_p^*(F)$. As a consequence, we require just a single propositional variable to encode formula F_{eg}^*.

As a further refinement, we could apply methods devised by Pnueli *et al.* to reduce the size of the domains associated with each variable in $\Sigma_g^*(F)$ [PRSS99]. This will in turn allow us to reduce the number of propositional variables required to encode each domain variable in $\Sigma_g^*(F)$.

5.2 Translation Based on Pairwise Encodings of Term Equality

Goel *et al.* [GSZAS98] describe a method for generating a propositional formula from an EUF formula, such that the propositional formula will be a tautology if and only if the EUF formula is universally valid. They first use Ackermann's method to eliminate function applications of nonzero order [Ack54]. Then they introduce a propositional variable $e_{i,j}$ for each pair of domain variables v_i and v_j encoding the conditions under which the two variables have matching values. Finally, they generate a propositional formula in terms of the $e_{i,j}$ variables.

The propositional formula they generate does not enforce constraints among the $e_{i,j}$ variables due to the transitivity of equality, i.e., constraints of the form $e_{i,j} \wedge e_{j,k} \Rightarrow e_{i,k}$. As a result, in attempting to prove the formula is a tautology, they may generate false "counterexamples." They express the set of potential counterexamples as a BDD and then systematically eliminate those that contain transitivity violations.

We provide a modified formulation of their approach that exploits the properties of p-formulas to encode only valuations under maximally diverse interpretations. As a consequence, we require $e_{i,j}$ variables only to express equality among those domain variables that represent g-term values in the original formula.

We describe a method of expressing the transitivity constraints in our formulas that exploits the sparse structure of the $e_{i,j}$ variables. In practice, we have actually found that our processor models can be verified without enforcing any transitivity constraints. Apparently the transitivity conditions that caused problems for Goel *et al.* correspond to p-terms in our verifications and hence do not require any propositional variables.

5.2.1 Construction of Propositional Formula

Starting with p-formula F, we apply our method of eliminating function applications to give a formula F^* containing only domain and propositional variables. The domain variables in F^* are partitioned into sets $\Sigma_p^*(F)$, corresponding to p-function applications in F, and $\Sigma_g^*(F)$ corresponding to g-function applications in F. Let us identify the variables in $\Sigma_g^*(F)$ as $\{v_1, \ldots, v_N\}$, and the variables in $\Sigma_p^*(F)$ as $\{v_{N+1}, \ldots, v_{N+M}\}$. We need only encode interpretations that are diverse in this latter set of variables.

For values of i and j such that $1 \leq i < j \leq N$, define propositional variables $e_{i,j}$ encoding the equality relation between variables v_i and v_j. We require these propositional variables only for indices less than or equal to N. Higher indices correspond to variables in $\Sigma_p^*(F)$, and we can assume for any such variable v_i that it will equal variable v_j only when $i = j$.

For each term T in F^*, and each v_i with $1 \leq i \leq N + M$, we generate formulas of the form $enct_i(T)$ for $1 \leq i \leq N + M$ to encode the conditions under which the control formulas in the *ITE*s in term T will be set so that value of T becomes that of domain variable v_i. In addition, for each formula G we define a propositional formula $encf(G)$ giving the encoded form of G. These formulas are defined by mutual recursion. The base cases are:

$$\begin{array}{rcll} \mathit{encf}(\mathbf{true}) & \doteq & \mathbf{true} & \\ \mathit{encf}(\mathbf{false}) & \doteq & \mathbf{false} & \\ \mathit{encf}(a) & \doteq & a, & a \text{ is a propositional variable} \\ \mathit{enct}_i(v_i) & \doteq & \mathbf{true} & \\ \mathit{enct}_j(v_i) & \doteq & \mathbf{false}, & \text{For } i \neq j \end{array}$$

For the logical connectives, we define encf in the obvious way:

$$\begin{array}{rcl} \mathit{encf}(\neg G_1) & \doteq & \neg \mathit{encf}(G_1) \\ \mathit{encf}(G_1 \wedge G_2) & \doteq & \mathit{encf}(G_1) \wedge \mathit{encf}(G_2) \\ \mathit{encf}(G_1 \vee G_2) & \doteq & \mathit{encf}(G_1) \vee \mathit{encf}(G_2) \end{array}$$

For *ITE* terms, we define enct as:

$$\mathit{enct}_i(\mathit{ITE}(G, T_1, T_2)) \quad \doteq \quad \mathit{encf}(G) \wedge \mathit{enct}_i(T_1) \quad \vee \quad \neg \mathit{encf}(G) \wedge \mathit{enct}_i(T_2)$$

For equations, we define $\mathit{encf}(T_1 = T_2)$ to be

$$\mathit{encf}(T_1 = T_2) \quad \doteq \quad \bigvee_{1 \leq i,j \leq N} \mathit{enct}_i(T_1) \wedge e_{[i,j]} \wedge \mathit{enct}_j(T_2) \quad \vee \quad \bigvee_{N+1 \leq i \leq N+M} \mathit{enct}_i(T_1) \wedge \mathit{enct}_i(T_2) \tag{6}$$

where $e_{[i,j]}$ is defined for $1 \leq i, j \leq N$ as:

$$e_{[i,j]} \quad \doteq \quad \begin{cases} \mathbf{true} & i = j \\ e_{i,j} & i < j \\ e_{j,i} & i > j \end{cases}$$

Informally, Equation 6 expresses the property that there are two ways for a pair of terms to be equal in an interpretation. The first way is if the two terms evaluate to the same variable, i.e., we have both $\mathit{enct}_i(T_1)$ and $\mathit{enct}_i(T_2)$ hold for some variable v_i. For $1 \leq i \leq N$, the left hand part of Equation 6 will hold since $e_{[i,i]} = \mathbf{true}$. For $N + 1 \leq i \leq N$, the right hand part of Equation 6 will hold. The second way is that two terms will be equal under some interpretation when they evaluate to two different variables v_i and v_j that have the same value. In this case we will have $\mathit{enct}_i(T_1)$, $\mathit{enct}_j(T_2)$, and $e_{[i,j]}$ hold, where $1 \leq i, j \leq N$. Observe that Equation 6 encodes only interpretations that are diverse over $\{v_{N+1}, \ldots, v_{N+M}\}$. It makes use of the fact that when $N + 1 \leq i \leq N + M$, variable v_i will only equal variable v_j only if $i = j$.

As an example, Figure 7 shows an encoding of formula F^* given in Figure 4, which was derived from the original formula F shown in Figure 3. The variables in $\Sigma_g^*(F^*)$ are x and y. These are renamed as v_1 and v_2, giving $N = 2$. The variables in $\Sigma_p^*(F^*)$ are vg_1, vg_2, vg_3, vh_1, and

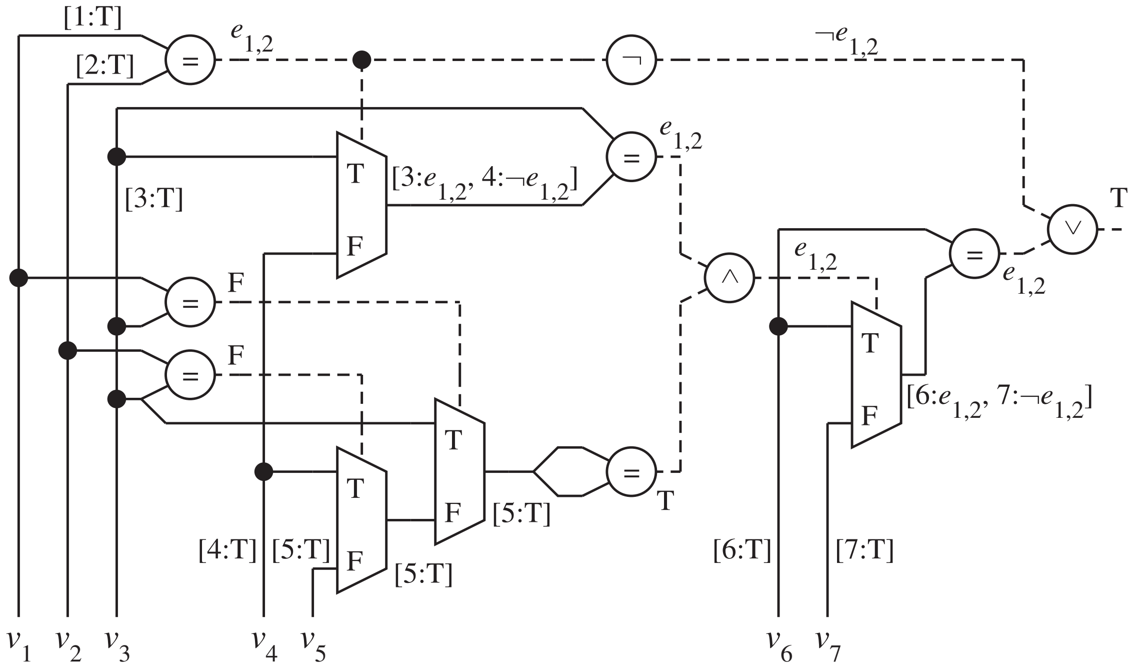


Figure 7: Encoding Example Formula in Propositional Logic. Each term T is represented as a list giving the non-**false** values of $enct_i(T)$.

vh_2. These are relabeled as v_3 through v_7, giving $M = 5$. Each formula in the figure is annotated by a (simplified) propositional formula, while each term T is annotated by a list with entries of the form i: $enct_i(T)$, for those entries such that $enct_i(T) \neq \mathbf{false}$. We use the shorthand notation "T" for $\mathbf{true}$ and "F" for $\mathbf{false}$. Our encoding introduces a single propositional variable $e_{1,2}$. It can be seen that our method encodes only the interpretations for F^* labeled as D1 and D2 in Table 2. When $e_{1,2}$ is false, we encode interpretation D2, in which $x \neq y$ and every function application term yields a distinct value. When $e_{1,2}$ is true, we encode interpretation D1, in which $x = y$ and hence we have $g(x) = g(y)$ and $h(g(x), g(g(x))) = h(g(y), g(g(y)))$.

In general, the final result of the recursive translation will be a propositional formula $encf(F^*)$. The variables in this formula consist of the propositional variables that occur in F^* as well as a subset of the variables of the form $e_{i,j}$. Nothing in this formula enforces the transitivity of equality. We will discuss in the next section how to impose transitivity constraints in a way that exploits the sparse structure of the equations. Other than transitivity, we claim that the translation $encf(F^*)$ captures validity of F^*, and consequently the original p-formula F. For an interpretation J over a set of propositional variables, including variables of the form $e_{i,j}$ for $1 \leq i < j \leq N$, we say that J *obeys transitivity* when for all i, j, and k such that $1 \leq i, j, k \leq N$ we have $J[e_{[i,j]}] \wedge J[e_{[j,k]}] \Rightarrow J[e_{[i,k]}]$.

To formalize the intuition behind the encoding, let I^* be an interpretation of the variables in the translated formula F^*. For interpretation I^*, define $sel_{I^*}(T)$ to be a function mapping each term T in F^* to the index of the unique domain variable selected by the values of the *ITE* control formulas in T. That is, $sel_{I^*}(v_i) \doteq i$, while $sel_{I^*}(ITE(G, T_1, T_2))$ is defined as $sel_{I^*}(T_1)$ when $I^*[G] = \mathbf{true}$ and as $sel_{I^*}(T_2)$ when $I^*[G] = \mathbf{false}$.

Proposition 2 *For all interpretations I^* of the variables in F^* and any term T occurring in F^*, if $sel_{I^*}(T) = i$, then $I^*[T] = I^*(v_i)$.*

Lemma 8 *For any interpretation I^* of the variables in F^* that is diverse for $\Sigma_p^*(F)$, there is an interpretation J of the variables in $encf(F^*)$ that obeys transitivity and such that $J[encf(F^*)] = I^*[F^*]$.*

Proof: For each propositional variable a occurring in F^*, we define $J(a) \doteq I^*(a)$. For each pair of variables v_i and v_j such that $1 \leq i < j \leq N$, we define $J(e_{i,j})$ to be $\mathbf{true}$ iff $I^*(v_i) = I^*(v_j)$. We can see that J must obey transitivity, because it is defined in terms of a transitive relation in I^*.

We prove the following hypothesis by induction on the expression depths:

1. For every formula G in F^*: $J[encf(G)] = I^*[G]$.
2. For every term T in F^* and all i such that $1 \leq i \leq N + M$: $J[enct_i(T)] = \mathbf{true}$ iff $sel_{I^*}(T) = i$.

The base cases hold as follows:

1. Formulas of the form $\mathbf{true}$, $\mathbf{false}$, and a have $encf(G) = G$ and $J[G] = I^*[G]$.
2. Term v_j has $J[enct_i(v_j)] = \mathbf{true}$ iff $j = i$, and $sel_{I^*}(v_j) = i$ iff $j = i$.

Assuming the induction hypothesis holds for formulas G_1 and G_2, one can readily see that it will hold for formulas $\neg G_1$, $G_1 \wedge G_2$, and $G_1 \vee G_2$, by the definition of $encf$

Assuming the induction hypothesis holds for formula G and for terms T_1 and T_2, consider term T of the form *ITE*(G, T_1, T_2). For the case where $I^*[G] = \mathbf{true}$, we have $I^*[T] = I^*[T_1]$, and also $sel_{I^*}(T) = sel_{I^*}(T_1)$. The induction hypotheses for T_1 gives $J[enct_i(T_1)] = \mathbf{true}$ iff $sel_{I^*}(T_1) = i$. The induction hypothesis for G gives $J[encf(G)] = I^*[G] = \mathbf{true}$, and hence $J[enct_i(T)] = J[enct_i(T_1)]$. From all this, we can conclude that $J[enct_i(T)] = \mathbf{true}$ iff $sel_{I^*}(T) = i$. A similar argument holds when $I^*[G] = \mathbf{false}$, but based on the induction hypothesis for T_2.

Finally, assuming the induction hypothesis holds for terms T_1 and T_2, consider the equation $T_1 = T_2$. Suppose that $sel_{I^*}(T_1) = i$ and $sel_{I^*}(T_2) = j$. Our induction hypothesis for T_1 and T_2 give $J[enct_i(T_1)] = J[enct_j(T_2)] = \mathbf{true}$. Suppose either $i > N$ or $j > N$. Then we will have $I^*(v_i) = I^*(v_j)$ iff $i = j$. In addition, the right hand part of Equation 6 will hold under J iff $i = j$. Otherwise, suppose that $1 \leq i, j \leq N$. We will have $I^*(v_i) = I^*(v_j)$ iff $J[e_{[i,j]}] = \mathbf{true}$. In addition, the left hand part of Equation 6 will hold under J iff $J[e_{[i,j]}] = \mathbf{true}$ $\Box$

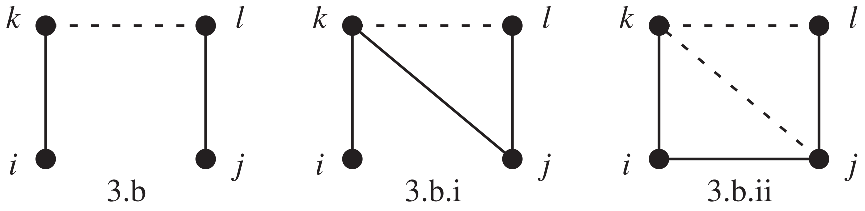


Figure 8: Case Analysis for Part 3b of Proof of Lemma 9. Solid lines denote equalities, while dashed lines denote inequalities.

Lemma 9 *For every interpretation J of the variables in $encf(F^*)$ that obeys transitivity, there is an interpretation I^* of the variables in F^* such that $I[F^*] = J[encf(F^*)]$.*

Proof: We define interpretation I^* over the domain of integers $\{1, \ldots, N + M\}$. For propositional variable a, we define $I^*(a) = J(a)$. For $1 \leq j \leq N$ we let $I^*(v_j)$ be the minimum value of i such that $J[e_{[i,j]}] = \mathbf{true}$. For $N < j \leq N + M$ we let $I^*(v_j) = j$. Observe that this interpretation gives $I^*(v_j) \leq j$ for all $j \leq N$, since $e_{[j,j]} = \mathbf{true}$, and $I^*(v_j) = j$ for $j > N$.

We claim that for $i \leq N$, if $I^*(v_j) = i$, then we must have $I^*(v_i) = i$ as well. If instead we had $I^*(v_i) = k < i$, then we must have $J[e_{[k,i]}] = \mathbf{true}$. Combining this with $J[e_{[i,j]}] = \mathbf{true}$, the transitivity requirement would give $J[e_{[k,j]}] = \mathbf{true}$, but this would imply that $I^*(v_j) = k \neq i$.

We prove the following hypothesis by induction on the expression depths:

1. For every formula G in F^*: $I^*[G] = J[encf(G)]$.

2. For every term T in F^* and all i such that $1 \leq i \leq N + M$: $sel_{I^*}(T) = i$ iff $J[enct_i(T)] = \mathbf{true}$.

The base cases hold as follows:

1. Formulas of the form $\mathbf{true}$, $\mathbf{false}$, and a have $G = encf(G)$ and $I^*[G] = J[G]$.

2. Term v_j has $sel_{I^*}(v_j) = i$ iff $j = i$ and $J[enct_i(v_j)] = \mathbf{true}$ iff $j = i$.

Assuming the induction hypothesis holds for formula G and for terms T_1 and T_2, consider term T of the form *ITE*(G, T_1, T_2). For the case where $J[encf(G)] = \mathbf{true}$, we have $J[enct_i(T)] = J[enct_i(T_1)]$. The induction hypothesis for T_1 gives $sel_{I^*}(T_1) = i$ iff $J[enct_i(T_1)] = \mathbf{true}$. The induction hypothesis for G gives $I^*[G] = J[encf(G)] = \mathbf{true}$, giving $I^*[T] = I^*[T_1]$, and also $sel_{I^*}(T) = sel_{I^*}(T_1)$. Combining all his gives $sel_{I^*}(T) = i$ iff $J[enct_i(T)] = \mathbf{true}$. A similar argument can be made when $J[encf(G)] = \mathbf{false}$, but based on the induction hypothesis for T_2.

Finally, assuming the induction hypothesis holds for terms T_1 and T_2, consider the equation $T_1 = T_2$. Let $i = sel_{I^*}(T_1)$ and $j = sel_{I^*}(T_2)$. In addition, let $k = I^*(v_i)$ and $l = I^*(v_j)$. Our induction hypothesis gives $J[enct_i(T_1)] = \mathtt{true}$, and $J[enct_j(T_2)] = \mathtt{true}$. Proposition 2 gives $I^*[T_1] = k$ and $I^*[T_2] = l$. By our earlier argument, we must also have $I^*(v_k) = k$ and $I^*(v_l) = l$. We consider different cases for the values of i, j, k, and l.

1. Suppose $i > N$. Then we must have $k = I^*(v_i) = i$. Equation $T_1 = T_2$ will hold under I^* iff $I^*(v_j) = l = k$, and this will hold iff $j = l = k = i$. In addition, the right hand part of Equation 6 will hold under J iff $i = j$.
2. Suppose $j > N$. By an argument similar to the previous one, we will have equation $T_1 = T_2$ holding under interpretation I^* and Equation 6 holding under interpretation J iff $i = j$.
3. Suppose $1 \leq i, j \leq N$. Since $I^*(v_i) = k = I^*(v_k)$ we must have $J[e_{[k,i]}] = \mathtt{true}$. Similarly, since $I^*(v_j) = l = I^*(v_l)$ we must have $J[e_{[l,j]}] = \mathtt{true}$.
   (a) Suppose $k = l$, and hence $T_1 = T_2$ holds under I^*. Then we have $J[e_{[i,k]}] = J[e_{[k,j]}] = \mathtt{true}$. Our transitivity requirement then gives $J[e_{[i,j]}] = \mathtt{true}$, and hence the left hand part of Equation 6 will hold under J.
   (b) Suppose $k \neq l$, and hence $T_1 = T_2$ does not hold under I^*. We must have $J[e_{[k,l]}] = \mathtt{false}$. This condition is illustrated in the left hand diagram of Figure 8. In this figure we use solid lines to denote equalities and dashed lines to denote inequalities. We argue that we must also have $J[e_{[i,j]}] = \mathtt{false}$ by the following case analysis for $e_{[k,j]}$:
      i. For $J[e_{[k,j]}] = \mathtt{true}$, we get the case diagrammed in the middle of Figure 8 where the diagonal line creates a triangle with just one dashed line (inequality). This represents a violation of our transitivity requirement, since it indicates $J[e_{[k,j]}] = J[e_{[j,l]}] = \mathtt{true}$, but $J[e_{[k,l]}] = \mathtt{false}$.
      ii. For $J[e_{[k,j]}] = \mathtt{false}$ and $J[e_{[i,j]}] = \mathtt{true}$, we have the case diagrammed on the right side of Figure 8. Again we have a triangle with just one dashed line indicating a violation of our transitivity requirement, with $J[e_{[k,i]}] = J[e_{[i,j]}] = \mathtt{true}$, but $J[e_{[k,j]}] = \mathtt{false}$.

      With $J[e_{[i,j]}] = \mathtt{false}$, Equation 6 will not hold under J.

From this case analysis we see that $T_1 = T_2$ holds under I^* iff Equation 6 holds under J. □

5.2.2 Transitivity Constraints

We may need to constrain our top level formula to only consider interpretations of the variables of the form $e_{i,j}$ that preserve the transitivity of equality. For example, if we have variables $e_{1,2}$, $e_{2,3}$, and $e_{1,3}$, we want to avoid interpretations that assign values $\mathtt{true}$ to two of these variables, but $\mathtt{false}$ to the third. On the other hand, there is no need add transitivity constraints for cases where the equality of two subexpressions has no bearing on the truth of our top-level formula.

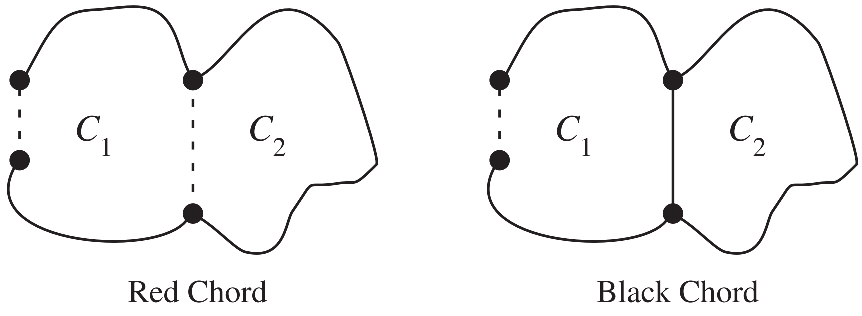


Figure 9: Case Analysis for Proof of Lemma 10. Solid lines denote black edges (equalities), while dashed lines denote red edges (inequalities).

We therefore propose a method of enforcing transitivity that exploits the sparse structure of the equality comparisons. We view this task as one of generating a set of constraints $Trans$, where each constraint is a formula over the $e_{i,j}$ variables. Our final verification condition is then expressed as the formula $[\bigwedge_{G \in Trans} G] \Rightarrow encf(F^*)$.

Let X denote the set of all variables of the form $e_{i,j}$ occurring in $encf(F^*)$. Create an undirected graph having a vertex for every i such that $1 \leq i \leq N$, and an edge (i, j) for every variable of the form $e_{i,j}$ in X. For an interpretation J of the variables in X, color edge (i, j) red when $J(x_{i,j})$ is $\mathbf{false}$ and color it black when $J(x_{i,j})$ is $\mathtt{true}$. One can see that this interpretation will violate transitivity if and only if there is some cycle in the graph containing exactly one red edge. This generalizes the case for triangles we saw in Figure 8, where red edges are denoted with dashed lines. We must add constraints to $Trans$ that eliminate such interpretations.

Rather than enumerating all of the cycles in the graph, we augment the set X with additional variables of the form $e_{i,j}$ such that the resulting graph becomes *chordal.* [Rose70]. That is, the graph has the property that for every cycle of length greater than 3, there is an edge (called a *chord* of the cycle) connecting two vertices that are not adjacent in the cycle. Such graphs have been studied extensively in the context of sparse Gaussian elimination. In fact, the problem of finding a minimum set of additional variables to add to our set is identical to the problem of finding an elimination ordering for Gaussian elimination that minimizes the amount of fill-in. Although this problem is NP-complete [Yan81], there are good heuristic solutions.

Lemma 10 *If a chordal graph contains no triangle having exactly one red edge, then it contains no cycles containing exactly one red edge.*

Proof: The proof proceeds by induction on the cycle length, with cycles of length 3 forming the trivial base case. Assume some cycle C of length k greater than 3 contains exactly one red edge, but no smaller cycles have this property. Cycle C must have a chord splitting it into two cycles C_1 and C_2, both of which are smaller than k, and both containing the chord. Assume without loss of generality that the red edge of C is in C_1. Consider the two cases illustrated in Figure 9. If the chord is colored red (left), this would be the only red edge in cycle C_2. If the chord is colored black (right), then cycle C_1 would contain the only red edge that occurs in C. In either case, we have found a cycle of length less than k containing exactly one red edge, contradicting our assumption about C. $\Box$

Assume this augmentation yields a set of variables X'. Then for every value of i, j, and k, such that $i < j$ and $j < k$, and such that there are variables $e_{i,j}$, $e_{i,k}$, and $e_{j,k}$ in X', we add three transitivity constraints to $Trans$: $e_{i,j} \wedge e_{j,k} \Rightarrow e_{i,k}$, $e_{i,k} \wedge e_{j,k} \Rightarrow e_{i,j}$, and $e_{i,j} \wedge e_{i,k} \Rightarrow e_{j,k}$. These constraints guarantee that any interpretation of the variables in X' gives an edge coloring that has no cycle of length 3 containing exactly one red edge. By Lemma 10 this property guarantees that no larger cycle can have exactly one red edge, either, and hence the interpretation must satisfy transitivity.

Theorem 4 *P-formula F is universally valid iff the propositional formula $[\bigwedge_{G \in Trans} G] \Rightarrow encf(F^*)$ is a tautology.*

Proof: This theorem follows directly from Lemmas 6, 7, and 10. $\Box$

As mentioned earlier, we have found in practice that we can verify our microprocessor designs without enforcing any transitivity constraints. The soundness of this optimization can be expressed as follows:

Corollary 2 *If propositional formula $[\bigwedge_{G \in Trans'} G] \Rightarrow encf(F^*)$ is a tautology for some $Trans' \subseteq Trans$, then p-formula F is universally valid.*

5.2.3 Discussion

In the formulation by Goel *et al.*, a propositional variable would be required for every pair of function applications occurring in the original formula. In our case, we need only introduce these variables for a subset of the pairs of g-function applications. For example, their method would require 8 variables to encode the transformed version of formula F_{eg} shown in Figure 6, whereas we require only 1 using either of our two encoding schemes. In addition, they found that adding transitivity constraints to the propositional formula directly caused a blow-up of the BDDs when evaluating the formula. In our case, we have far fewer variables, and we have proposed an approach to add only a minimal number of additional variables and transitivity constraints.

6 Modeling Microprocessors in PEUF

Our interest is in verifying pipelined microprocessors, proving their equivalence to an unpipelined instruction set architecture model. We use the approach pioneered by Burch and Dill [BD94] in which the abstraction function from pipeline state to architectural state is computed by symbolically simulating a flushing of the pipeline state and then projecting away the state of all but the architectural state elements, such as the register file, program counter, and data memory. Operationally, we construct two sets of p-terms describing the final values of the state elements resulting from two different symbolic simulation sequences—one from the pipeline model and one from the instruction set model. The correctness condition is represented by a p-formula expressing the equality of these two sets of p-terms.

Our approach starts with an RTL or gate-level model of the microprocessor and performs a series of abstractions to create a model of the data path using terms that satisfy the restrictions of PEUF. Examining the structure of a pipelined processor, we find that the signals we wish to abstract as terms can be classified as follows:

Program Data: Values generated by the ALU and stored in registers and data memory. These are also used as addresses for the data memory.

Register Identifiers: Used to index the register file

Instruction Addresses: Used to designate which instructions to fetch

Control values: Status flags, opcodes, and other signals modeled at the bit level.

By proper construction of the data path model, both program data and instruction addresses can be represented as p-terms. Register identifiers, on the other hand, must be modeled as g-terms, because their comparisons control the stall and bypass logic. The remaining control logic is kept at the bit level.

In order to generate such a model, we must abstract the operation of some of the processor units. For example, the data path ALU is abstracted as an uninterpreted p-function, generating a data value given its data and control inputs. Formally, this requires extending the syntax for function applications to allow both formula and term inputs. We model the PC incrementer and the branch target logic as uninterpreted functions generating instruction addresses. We model the branch decision logic as an uninterpreted predicate indicating whether or not to take the branch based on data and control inputs. This allows us to abstract away the data equality test used by the branch-on-equal instruction.

To model the register file, we use the memory model described by Burch and Dill [BD94], creating a nested *ITE* structure to encode the effect of a read operation based on the history of writes to the memory. That is, suppose at some point we have performed k write operations with addresses given by terms $A_1, \ldots, A_k$ and data given by terms $D_1, \ldots, D_k$. Then the effect of a read with address term A is a the term:

$$\mathit{ITE}(A = A_k, D_k, \mathit{ITE}(A = A_{k-1}, D_{k-1}, \cdots \mathit{ITE}(A = A_1, D_1, f_I(A)) \cdots)) \qquad (7)$$

where f_I is an uninterpreted function expressing the initial memory state. Note that the presence of these comparison and *ITE* operations requires register identifiers to be modeled with g-terms.

Since we view the instruction memory as being read-only, we can model the instruction memory as a collection of uninterpreted functions and predicates—each generating a different portion of the instruction field. Some of these will be p-functions (for generating immediate data), some will be g-functions (for generating register identifiers), and some will be predicates (for generating the different bits of the opcode). In practice, the interpretation of different portions of an instruction word depends on the instruction type, essentially forming a "tagged union" data type. Extracting and interpreting the different instruction fields during processor verification is an interesting research problem, but it lies outside the scope of this paper.

The data memory provides a greater modeling challenge. Since the memory addresses are generated by the ALU, they are considered program data, which we would like to model as p-terms. However, using a memory model similar to that used for the register file requires comparisons between addresses and *ITE* operations having the comparison results as control. Instead, we must create a more abstract memory model that weakens the semantics of a true memory to satisfy the restrictions of PEUF. Our abstraction models a memory as a generic state machine, computing a new state for each write operation based on the input data, address, and current state. Rather than Equation 7, we would express the effect of a read with address term A after k write operations as $f_r(S_k, A)$, where f_r is an uninterpreted "memory read" function, and S_k is a term representing the state of the memory after the k write operations. This term is defined recursively as $S_0 = s_0$, where s_0 is a domain variable representing the initial state, and $S_i = f_u(S_{i-1}, A_i, D_i)$ for $i \geq 1$, where f_u is an uninterpreted "memory update" function. In essence, we view write operations as making arbitrary changes to the entire memory state.

This model removes some of the correlations guaranteed by the read operations of an actual memory. For example, although it will yield identical operations for two successive read operations to the same address, it will indicate that possibly different result could be returned if these two reads are separated by a write, even to a different address. In addition, if we write data D to address A and then immediately read from this address, our model will not indicate that the resulting value must be D. Nonetheless, it can readily be seen that this abstraction is a conservative approximation of an actual memory. As long as the pipelined processor performs only the write operations indicated by the program, that it performs writes in program order, and that the ordering of reads relative to writes matches the program order, the two simulations will produce equal terms representing the final memory states.

The remaining parts of the data path include comparators comparing for matching register identifiers to determine bypass and stall conditions, and multiplexors, modeled as *ITE* operations selecting between alternate data and instruction address sources. Since register identifiers are modeled as g-terms, these comparison and control combinations obey the restrictions of PEUF. Finally, such operations as instruction decoding and pipeline control are modeled at the bit level using Boolean operations.

7 Experimental Results

In [VB98], we described the implementation of a symbolic simulator for verifying pipelined systems using vectors of Boolean variables to encode domain variables, effectively treating all terms as g-terms. This simulation is performed directly on a modified gate-level representation of the processor. In this modified version, we replace all state holding elements (registers, memories, and latches) with behavioral models we call Efficient Memory Models (EMMs). In addition all data-transformation elements (e.g., ALUs, shifters, PC incrementers) are replaced by read-only EMMs, which effectively implement the transformation of function applications into nested *ITE* expressions described in Section 4.2. One interesting feature of this implementation is that our decision procedure is executed directly as part of the symbolic simulation. Whereas other implementations, including Burch and Dill's, first generate a formula and then decide its validity, our implementation generates and manipulates bit-vector representations of terms as the symbolic simulation proceeds. Modifying this program to exploit positive equality simply involves having the EMMs generate expressions containing fixed bit patterns rather than vectors of Boolean variables. All performance results presented here were measured on a 125 MHz Sun Microsystems SPARC-20.

We constructed several simple pipeline processor design based on the MIPS instruction set [KH92]. We abstract register identifiers as g-terms, and hence our verification covers all possible numbers of program registers including the 32 of the MIPS instruction set. The simplest version of the pipeline implements ten different Register-Register and Register-Immediate instructions. Our program could verify this design in 48 seconds of CPU time and just 7 MB of memory using vectors of Boolean variables to encode domain variables. Using fixed bit patterns reduces the complexity of the verification to 6 seconds and 2 MB.

We then added a memory stage to implement load and store instructions. An interlock stalls the processor one cycle when a load instruction is followed by an instruction requiring the loaded result. Treating all terms as g-terms and using vectors of Boolean variables to encode domain variables, we could not verify even a 4-bit version of this data path (effectively reducing $|\mathcal{D}|$ to 16), despite running for over 2000 seconds. The fact that both addresses and data for the memory come from the register file induces a circular constraint on the ordering of BDD variables encoding the terms. On the other hand, exploiting positive equality by using fixed bit patterns for register values eliminates these variable ordering concerns. As a consequence, we could verify this design in just 12 CPU seconds using 1.8 MB.

Finally, we verified a complete CPU, with a 5-stage pipeline implementing 10 ALU instructions, load and store, and MIPS instructions `j` (jump with target computed from instruction word), `jr` (jump using register value as target), and `beq` (branch on equal). This design is comparable to the DLX design [HP96] verified by Burch and Dill in [BD94], although our version contains more of the implementation details. We were unable to verify this processor using the scheme of [VB98]. Having instruction addresses dependent on instruction or data values leads to exponential BDD growth when modeling the instruction memory. Modeling instruction addresses as p-terms, on the other hand, makes this verification tractable. We can verify the full, 32-bit version of the processor using 169 CPU seconds and 7.5 MB.

8 Conclusions

Eliminating Boolean variables in the encoding of terms representing program data and instruction addresses has given us a major breakthrough in our ability to verify pipelined processors. Our BDD variables now only encode control conditions and register identifiers. For classic RISC pipelines, the resulting state space is small and regular enough to be handled readily with BDDs.

We believe that there are many optimizations that will yield further improvements in the performance of Boolean methods for deciding formulas involving uninterpreted functions. We have found that relaxing functional consistency constraints to allow independent functionality of different instructions, as was done in [DPR98], can dramatically improve both memory and time performance. We look forward to testing our scheme for generating a propositional formula using Boolean variables to encode the relations between terms. Our method exploits positive equality to greatly reduce the number of propositional variables in the generated formula, as well as the number of functional consistency and transitivity constraints. We are also considering the use of satisfiability checkers rather than BDDs for performing our tautology checking

We consider pipelined processor verification to be a "grand challenge" problem for formal verification. We have found that complexity grows rapidly as we move to more complex pipelines, including ones with out-of-order execution and register renaming. Further breakthroughs will be required before we can handle complete models of state-of-the art processors.

References

[Ack54] W. Ackermann, *Solvable Cases of the Decision Problem*, North-Holland, Amsterdam, 1954.

[BDL96] C. Barrett, D. Dill, and J. Levitt, "Validity checking for combinations of theories with equality," *Formal Methods in Computer-Aided Design (FMCAD '96)*, M. Srivas and A. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November, 1996, pp. 187–201.

[BBCZ98] S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu, "Combining symbolic model checking with uninterpreted functions for out of order processor verification," *Formal Methods in Computer-Aided Design (FMCAD '98)*, G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November, 1998, pp. 187–201.

[Bry86] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.

[BD94] J. R. Burch, and D. L. Dill, "Automated verification of pipelined microprocessor control," *Computer-Aided Verification (CAV '94)*, D. L. Dill, *ed.*, LNCS 818, Springer-Verlag, June, 1994, pp. 68–80.

[Bur96] J. R. Burch, "Techniques for verifying superscalar microprocessors," *33rd Design Automation Conference (DAC '96)*, June, 1996, pp. 552–557.

[DPR98] W. Damm, A. Pnueli, and S. Ruah, "Herbrand automata for hardware verification," *9th International Conference on Concurrency Theory (CONCUR '98)*, Springer-Verlag, September, 1998.

[GJ79] M. R. Garey, and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, 1979.

[GSZAS98] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD based procedures for a theory of equality with uninterpreted functions," *Computer-Aided Verification (CAV '98)*, A. J. Hu and M. Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June, 1998, pp. 244–255.

[HP96] J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition Morgan-Kaufmann, San Francisco, 1996.

[HKGB97] R. Hojati, A. Kuehlmann, S. German, and R. K. Brayton, "Validity checking in the theory of equality with uinterpreted functions using finite instantiations," Unpublished paper presented at the *International Workshop on Logic Synthesis*, 1997.

[KH92] G. Kane, and J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.

[NO80] G. Nelson, and D. C. Oppen, "Fast decision procedures based on the congruence closure," *J. ACM*, Vol. 27, No. 2 (1980), pp. 356–364.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding equality formulas by small-domain instantiations," *Computer-Aided Verification CAV '99*, 1999.

[Rose70] Rose, D., "Triangulated graphs and the elimination process," *Journal of Mathematical Analysis and Applications*, Vol. 32 (1970), pp. 597–609.

[Sho79] R. E. Shostak, "A practical decision procedure for arithmetic with function symbols," *J. ACM*, Vol. 26, No. 2 (1979), pp. 351–360.

[VB98] M. N. Velev, and R. E. Bryant, "Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking." *Formal Methods in Computer-Aided Design (FMCAD '98)*, G. Gopalakrishnan and P. Windley, *eds.*, LNCS 1522, Springer-Verlag, November, 1998, pp. 18–35.

[Yan81] M. Yannakakis, "Computing the minimum fill-in is NP-complete," *SIAM Journal of Algebraic and Discrete Mathematics*, Vol. 2 (1981), pp. 77–79.