Implementability Among Predicates

Matthew Cook and Jehoshua Bruck
California Institute of Technology
Parallel and Distributed Computing Laboratory
Pasadena, California 91125
{cook,bruck}@paradise.caltech.edu

Abstract

Much work has been done to understand when given predicates (relations) on discrete variables can be conjoined to implement other predicates. Indeed, the lattice of "co-clones" (sets of predicates closed under conjunction, variable renaming, and existential quantification of variables) has been investigated steadily from the 1960's to the present. Here, we investigate a more general model, where duplicatability of values is not taken for granted. This model is motivated in part by large scale neural models, where duplicating a value is similar in cost to computing a function, and by quantum mechanics, where values cannot be duplicated. Implementations in this case are naturally given by a graph fragment in which vertices are predicates, internal edges are existentially quantified variables, and "dangling edges" (edges emanating from a vertex but not yet connected to another vertex) are the free variables of the implemented predicate. We examine questions of implementability among predicates in this scenario, and we present the solution to all implementability problems for single predicates on up to three boolean values. However, we find that a variety of proof methods are required, and the question of implementability indeed becomes undecidable for larger predicates, although this is tricky to prove. We find that most predicates cannot implement the 3-way equality predicate, which reaffirms the view that duplicatability of values should not be assumed a priori.

1 Predicate Graphs

This section will present our model for combining predicates. The next section will discuss how this model relates to some other common models, and the remaining sections will present our results.

By a predicate on n variables, we mean a relation

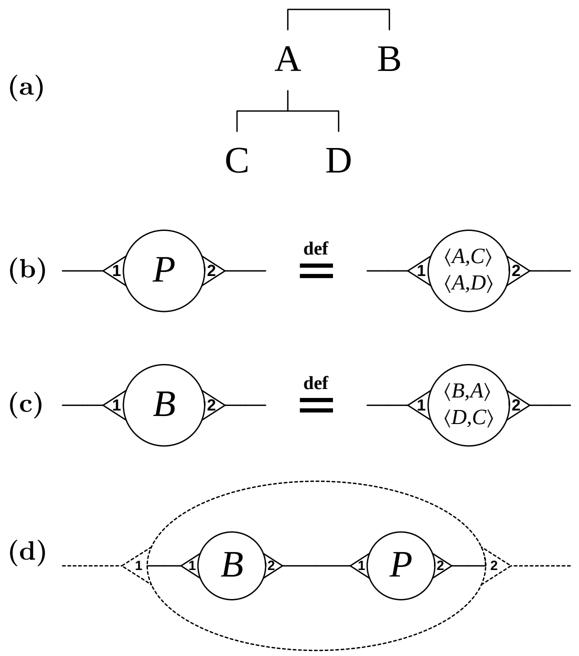


Figure 1. (a) A simple family tree for Alice (A), who has a brother Basil (B), and two children, Clara (C) and Desmond (D). **(b)** The *is-a-parent-of* predicate P is defined by the $\langle\text{parent}, \text{child}\rangle$ pairs that satisfy it. **(c)** The *is-a-brother-of* predicate B is defined similarly. **(d)** The two predicates can be combined so that the second variable of B is identified with the first variable of P. This implements the *is-an-uncle-of* predicate. A predicate graph such as this implementation is defined to be satisfied for a given set of dangling edge values iff appropriate internal edge values can then be chosen so that all the predicates in the graph are satisfied.

on n variables, that is, a set of ordered n-tuples, where each tuple indicates a valid combination of values for the n variables. For a predicate P, we may write $P(x, y, \ldots)$ to mean $\langle x, y, \ldots \rangle \in P$.

For example, say Alice and Basil are siblings, and Alice has children Clara and Desmond, as shown in Figure 1(a). We might define the predicate P (meaning *"is a parent of"*) to be $\{\langle \text{Alice}, \text{Clara} \rangle, \langle \text{Alice}, \text{Desmond} \rangle\}$. This is a list of all pairs $\langle p, c \rangle$, in our discrete population, where the first element p is the parent of the second element c. We can indicate P schematically as in Figure 1(b).

Similarly, we can define the predicate B (meaning *"is a brother of"*) to be $\{\langle \text{Basil}, \text{Alice} \rangle, \langle \text{Desmond}, \text{Clara} \rangle\}$, as in Figure 1(c).

We can then combine these predicates simply by saying that one of the variables used by one of them should be identified with one of the variables used by the other, as indicated in Figure 1(d), where we implement U (meaning *"is an uncle of"*) using a B and a P.

More formally, we define a *predicate graph* as a graph with a predicate at each vertex and a variable along each edge. Each edge is either an *internal edge* (connecting two vertices), or a *dangling edge* (a hyperedge connected to just a single vertex). At each vertex, the edges are ordered, so that the predicate at that vertex (which must be on n variables for a vertex of degree n) applies to the variables of the edges in the given order.

We allow more than one edge to connect a given pair of vertices, so long as each edge has a separate entry in the edge orderings at those vertices. Also, we allow edges to have both ends be at the same vertex, so long as each end has a separate entry in the edge ordering at that vertex. We will refer to such edges as *self-loops.* (If self-loops or multiple edges are objectionable, one can avoid them by inserting a vertex of degree 2 halfway along any objectionable edge, with an equality predicate.)

A predicate graph G with n (ordered) dangling edges is said to *implement* the predicate Q on n variables, where Q is those n-tuples of values for dangling-edge variables for which the conjunction of all of the vertex predicates on all the edge variables is satisfiable (i.e. true when all variables of internal edges are existentially quantified).

For example, in the implementation of Figure 1(d), U is implemented as:

$$U(x, z) = \exists y. [B(x, y) \wedge P(y, z)]$$

It should be clear that within a predicate graph, a single vertex and its predicate may be replaced by any implementation of that predicate, without affecting the satisfiability of the predicate graph.

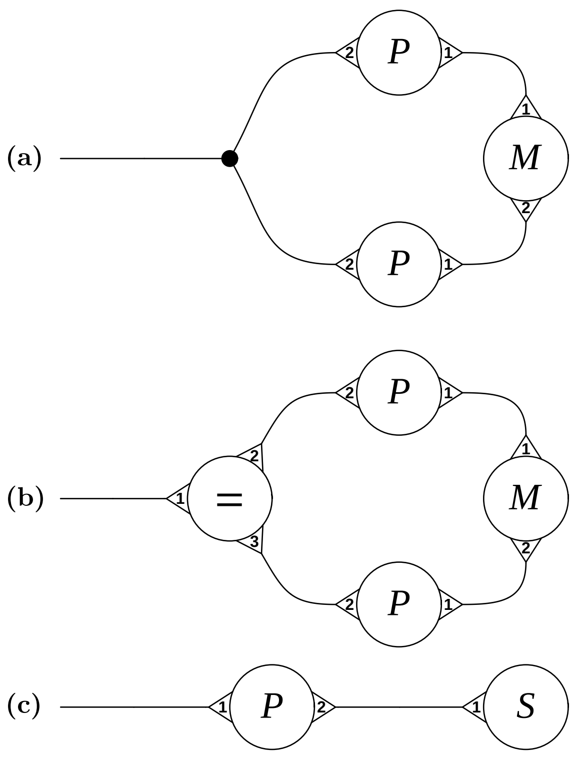


Figure 2. **(a)** A naive implementation of *has-equal-age-parents* in terms of *is-a-parent-of* (P) and *is-same-age-as* (M). **(b)** Here the dot is replaced with an equality predicate, so now this fits our definition of a predicate graph. **(c)** An implementation of the unary predicate *is-a-parent* in terms of *is-a-parent-of* (P) and *somebody* (S).

A key feature of this model is that information is localized. A variable's value is available only to the predicates at the two ends of the edge. This means that one variable cannot be used in many places, but rather in at most two places. (If it were limited to a single place, predicates would not be able to communicate at all.) If some information needs to be used many times, it must be explicitly duplicated, for example by a three-way equality predicate.

As an example, consider how one might implement the unary (one variable) predicate *"has equal age parents"*, using the parent predicate P and an *"is same age as"* predicate M. A potential solution is shown in Figure 2(a). However, there is something new here, a black dot connecting three edges, indicating that the edges all have the same value. How does this fit into our model? Quite simply, the black dot can be thought of as a vertex with the predicate *"all three are the same"*, better known as the equality predicate. This understanding is made explicit in Figure 2(b).

Note that since an internal edge must be of degree two, information cannot be destroyed (simply ignored), as every edge always leads to its other end. For example, to implement the unary *"is a parent"* predicate, one would need to use not only the predicate P, but also another predicate such as the trivial unary predicate S meaning *"somebody"*, defined here as $\{\langle\text{Alice}\rangle, \langle\text{Basil}\rangle, \langle\text{Clara}\rangle, \langle\text{Desmond}\rangle\}$, as shown in Figure 2(c). Intuitively, if we want to define *"is a parent"* in terms of *"is a parent of"*, then we have to say something like *"is a parent of somebody"*.

So we see that although at first our model looks restrictive, in that it may not be possible to duplicate or ignore values, in fact our model merely makes these capabilities explicit: The capability of duplicating values is embodied by availability of the three-way equality predicate, and the capability of ignoring values is embodied by availability of the unary predicate that accepts any value.

Is it possible to implement the *"has equal age parents"* predicate using any number of P, M, and S predicates, but without the equality predicate? This is the type of question that this paper will address. (We leave this particular example as a trivial exercise for the reader.)

2 Related Topics

This model is similar to several well-known models.

- Graph edge-coloring problems correspond to predicate graphs with no dangling edges, where each vertex has the *"all are different"* predicate.

- Constraint satisfaction problems (without optimization) [2, 4] can be viewed as bipartite predicate graphs where all the predicates in one of the parts are equality predicates (thus allowing variables to be used arbitrarily many times in the constraint satisfaction problem). Alternatively, one can view constraint satisfaction problems as predicate hypergraphs.

- Co-clones (also called relational clones) [1, 2, 6, 9], well known in the field of universal algebra, are sets of relations, containing the equality relation on two variables, which are closed under conjunction and existential quantification of variables. These correspond exactly to sets of predicates, containing the equality predicate on three variables, which are closed under predicate graph implementations.

- Tiling models such as Wang tiles [12] can be viewed as (perhaps infinite) predicate graphs with a specified graph topology (e.g. a square lattice), where every vertex has an identical predicate that accepts those combinations of values that correspond to one of the available tiles. The edge variables correspond to the markings on the edges of the tiles.

- Circuits of gates can be thought of as predicate graphs quite easily. Each gate can be thought of as a vertex whose predicate is the characteristic relation of the function computed by the gate. That is, the predicate accepts any combination of output and input values in which the gate's output value is the correct function of its input values. Instances of fan-out can be handled by the equality predicate as discussed above. The implemented predicate is then exactly the characteristic relation of the function implemented by the circuit of gates.

There are countless other areas where predicate graphs can arise. For example, the work that originally led us to consider this problem derived from a simple model of interacting neural assemblies, where each assembly implements a relation among the converging neural pathways [3].

3 The Lattice of Ternary Boolean Predicates

Here we consider the case of predicates on three Boolean variables, and ask when they can or can't implement each other.

Since implementability is transitive, and the empty graph is always implementable, and there exist universal sets of predicates (ones which can implement any other predicate), this means we can arrange all sets of predicates into a lattice based on implementability. We won't be able to examine the full lattice (it has an uncountable number of elements), nor can we reliably examine arbitrary parts of it (determining whether elements are comparable will turn out to be undecidable), but we can still examine the simplest parts of it.

Ternary (three-variable) Boolean predicates are the simplest predicates for which the implementability question can be interesting. Unary and binary predicates can only combine into graphs consisting of at most a single chain of predicates whose analysis at worst resembles the analysis of a finite state machine. Predicates on more than three variables, on the other hand, can always be recast as a tree of ternary predicates, if the arity of internal variables is not restricted.

Regarding the alphabet size, the Boolean case already exhibits considerable complexity, and of course predicates on larger alphabets can be recast as Boolean predicates if the number of variables is not restricted.

Figure 3 shows which ternary Boolean predicates can implement which others when negation (of a variable within such a predicate) is free. For example, the upper left oval is labeled "=1" to represent the predicate N that the sum of the values is 1, i.e. $N = \{\langle 0,0,1\rangle, \langle 0,1,0\rangle, \langle 1,0,0\rangle\}$. When we say that negation is free, we mean that the oval in fact represents not only the relation $N(x,y,z)$, but also the seven related relations, $N(\bar{x},y,z)$, $N(x,\bar{y},z)$, $N(\bar{x},\bar{y},z)$, etc., so the oval actually represents a set of eight relations. It is easy to see that if these eight relations can be used to implement another relation $T(x,y,z)$, then they can also implement $T(\bar{x},y,z)$ and so on, simply by negating the appropriate dangling edges of the implementation. So the notion of free negation is retained by implementations.

The motivations for considering free negation can be both external and internal to the theory of predicate graphs. The external motivation, which may or may not be present, is that the way in which predicates are connected together in a particular application may be such that negation is no harder than lack of negation. (For example, our original application transmitted the Boolean value along an edge of the graph via two wires, one of which could be "live" to indicate a value. In this case, crossing the two wires between predicates was no harder than not crossing them.) The internal motivation is that the results, in Figure 3, are much simpler to comprehend than the larger lattice of results without free negation, shown in Figure 4, and yet the overall structure of Figure 3 turns out to be a good first approximation to, and an aid in the understanding of, the structure of Figure 4.

Where one predicate is above another in the lattice, the upper one can implement the lower one. We use the term "lattice," even though there is not a unique top and bottom element, because Figures 3 and 4 are both subsets of the infinite lattice on all sets of Boolean predicates. The lack of a top element corresponds to the fact that there is no universal Boolean predicate (capable of implementing all others) on just three variables. Since graphs built with predicates which all use an even number of variables can only implement predicates on an even number of variables,[1] the smallest possible size for a universal Boolean predicate is five variables. There are many five-variable universal Boolean predicates, such as $\{\langle 1,0,0,0,0\rangle, \langle 0,1,1,0,0\rangle, \langle 0,0,0,1,1\rangle\}$. (Showing this is universal is a good homework problem.)

As an example implementation, the predicates $P_1 = \{\langle a,b,c\rangle \mid a \le b \le c\}$ and $P_2 = \{\langle a,b,c\rangle \mid a = b \le c\}$ can implement each other (this is left as an easy yet enjoyable exercise for the reader), so they share a single position in the lattice, marked as "2-SAT" (because deciding whether a graph of these predicates is satisfiable corresponds exactly to deciding a 2-SAT problem). In fact, this turns out to be one of only two cases in this lattice where two predicates can implement each other! The other case is for the position marked "0 / ☺", which stands for the two predicates which force one variable to a constant, leave another variable unrestricted (ignore it), and then either force or ignore the third variable.

It is interesting that in every case where one of the predicates in Figure 3 can implement another (and there are nearly 100 such cases), it can do so with a graph of just three predicates. (Three is the smallest possible size due to parity arguments.) We do not know of any simple reasoning that explains why this should be.

If we have OR$_3$ together with fan-out ($=_3$), we can implement any desired predicate by encoding a traditional 3-SAT representation of the predicate's characteristic function. (This is just one of many such small sets of predicates which are universal if used together.) So we see that it is possible to get a universal set of predicates even by combining predicates which, on their own, are not very powerful.

We can see that most small predicates cannot implement fan-out, indicating that fan-out is a valuable resource—if it is not directly available, it is unlikely to be constructible.

Figure 4 shows the full lattice for single predicates on three Boolean values. The predicates on two and zero values are not shown, but can easily be added underneath what is shown. Every predicate on one value is equivalent to the same predicate on three values. As we will discuss in the next section, a great number of proofs are involved in determining the lattices, as every connection or lack of connection is a theorem that must be proved.

Another natural question to ask is what the complexity is of simply deciding whether a given graph is satisfiable or not, for graphs built out of predicates at a given position in the lattice. The answer is that

[1] Recall that whether values can be ignored is determined by the availability of a predicate that ignores a single value. Such a predicate will happen to be implementable iff there is some non-empty predicate available on an odd number of variables and there is some predicate (perhaps the same one) available whose acceptable tuples do not all have the same parity. We will not present the construction that proves this, but a clever reader with a pencil can probably find it.

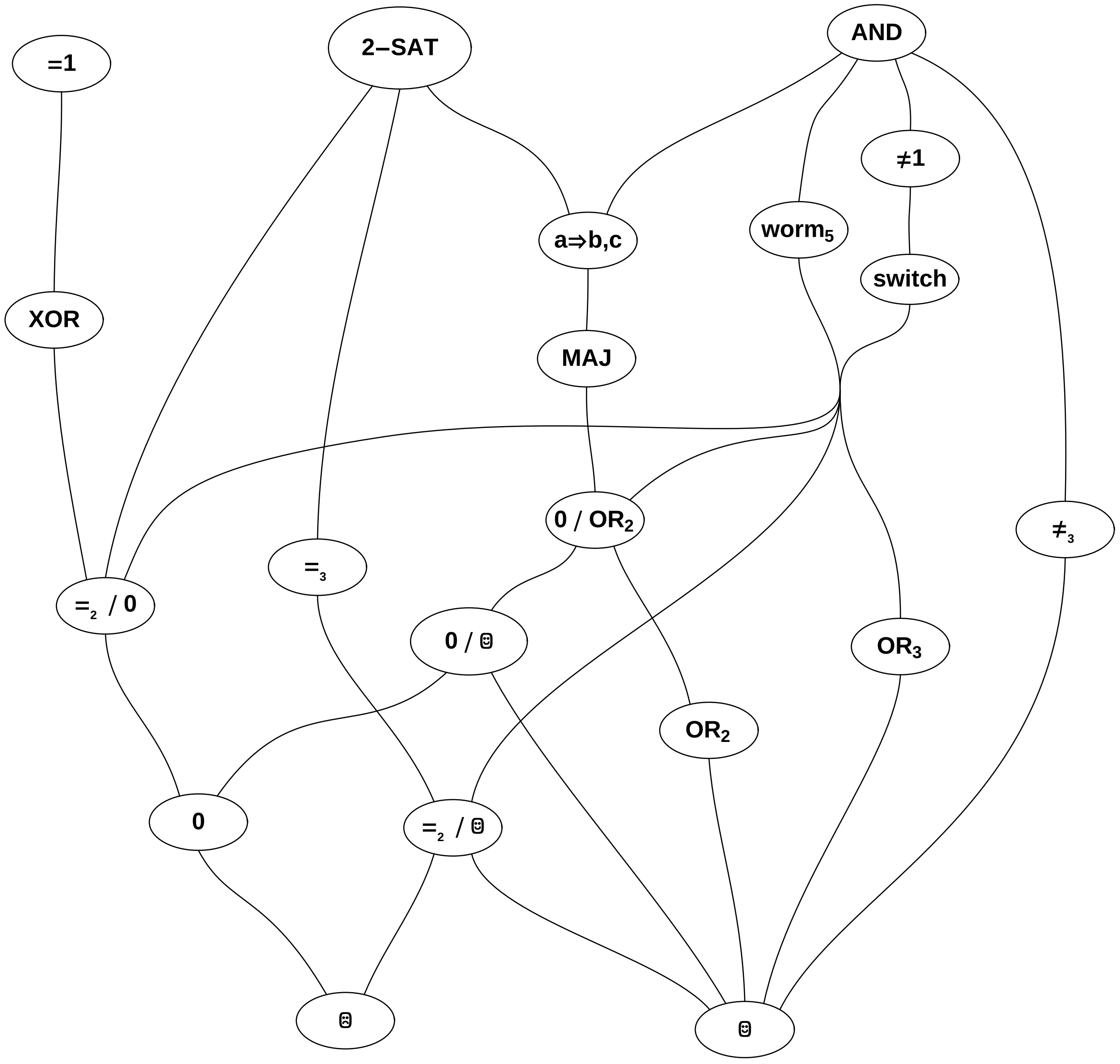


Figure 3. Which predicates can implement which others when variables can be negated. Where two ovals are connected by a line, the predicates represented by the upper oval are able to implement the predicates represented by the lower oval. Thus, the predicates at the top of this part of the lattice are the most powerful, and the ones at the bottom are the weakest. The meaning of the notations in the ovals is as follows: "$= 1$" accepts triples where $a + b + c = 1$. "XOR" accepts triples where $a + b + c \bmod 2 = 1$. "$=_n$" is the equality predicate on n variables. "0" is the *is-zero* predicate on a single variable. Where two notations appear in an oval, separated by a / between them, this indicates a predicate that contains both of the given predicates, operating independently on disjoint subsets of the variables. "☺" is the *don't-care* predicate that accepts all tuples. "☹" is the empty predicate that is always dissatisfied. "OR$_n$" is the predicate on n variables, that they are not all zero. "$\neq_3$" accepts triples where a, b, and c do not all have the same value. "MAJ" accepts triples where $a + b + c \geq 2$. "$a \Rightarrow b, c$" accepts triples where $a \Rightarrow b$ and $a \Rightarrow c$. "2-SAT" represents two predicates, one being $a \leq b \leq c$, and the other being $a = b \leq c$. "*worm*$_5$" represents the predicate that either $a \neq b$ or $a = b = c = 0$. "*switch*" represents $a \Rightarrow (b = c)$. "$\neq 1$" represents $a + b + c \neq 1$. "AND" represents $a = (b \wedge c)$.

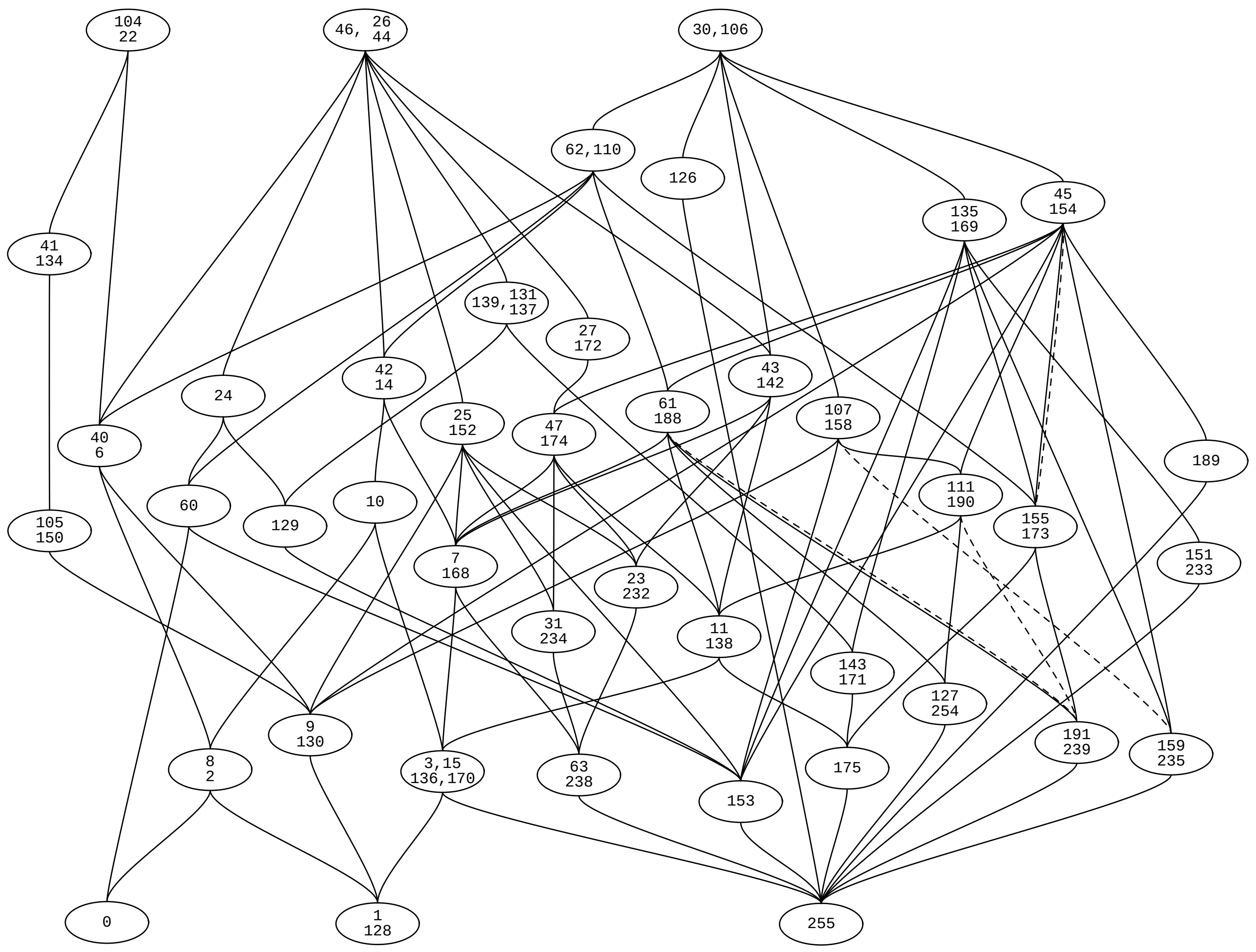


Figure 4. Which small predicates can implement which others. Each number indicates a predicate by treating each acceptable triple as a three digit binary number, and then calculating $\sum_{n \in triples} 2^n$. For example, the number 30 (in the top right oval) is $2^4 + 2^3 + 2^2 + 2^1$, so the triples accepted by that predicate are $\langle 1, 0, 0\rangle$, $\langle 0, 1, 1\rangle$, $\langle 0, 1, 0\rangle$, and $\langle 0, 0, 1\rangle$ (this is the predicate $A = \text{NOR}(B, C)$). Where two predicates can implement each other, they are shown in the same oval, separated by a comma. A surprising feature of the lattice is that this is not very common. If two predicates are dual to each other (by flipping 0's and 1's), then they are shown in the same oval, one above the other, in which case the oval really represents two distinct points in the lattice, one for each predicate. Lines between two such ovals only indicate implementability between the upper predicates in each oval, and between the lower predicates in each oval. A dotted line indicates crossing to the other side of the duality symmetry, so the upper predicate in the upper oval can implement the lower predicate in the lower oval, and the lower predicate in the upper oval can implement the upper predicate in the lower oval (these two implementations being equivalent by duality). Where three predicates are listed in an oval, the two atop each other are duals of each other, but each of the three can implement the other two. The oval containing four predicates represents two points in the lattice: one for the upper two predicates, and one for their duals, the lower two predicates.

only the AND predicate (the third peak) in Figure 3, and only the NAND/NOR predicates (the third peak) in Figure 4, are NP-complete; all of the other positions shown in Figures 3 and 4 (including those right below the third peak) can be decided in polynomial time. (An idea of Feder [5] shows that NAND is NP-complete. Schaeffer [11] has analyzed the case where fan-out is available.)

Apart from the symmetry of duality (which has been incorporated into the presentation of Figure 4 in order to simplify it), there do not appear to be any other significant symmetries or patterns. The edges of Figure 4 contain a lot of information, with little redundancy. In other words, when trying to prove whether or not a connection should be present in the lattice, looking at the structure of nearby connections does not provide any useful clues.

We can see that it is rarely the case that two predicates are equivalent in expressive power (i.e. that they occupy the same point in the lattice), but neither is it the case that every predicate is completely different from every other—there is significant height to the lattice, as well as width. In conclusion we observe that incomparability and comparability are both common for small predicates, but equivalence is rare.

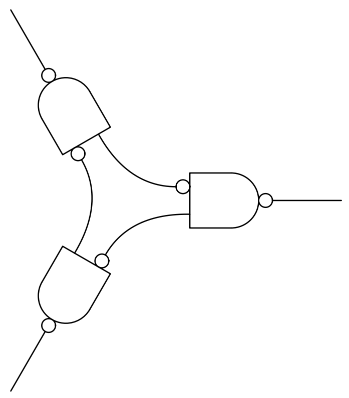


Figure 5. How AND implements MAJ. The AND predicate is represented using the traditional AND-gate icon. The circles at the ends of the edges indicate where an AND predicate uses a negated value of a variable. Since the negations happen to be used the same way in each predicate, this is also an example of an implementation without free negation, showing how predicate number 154 in Figure 4 can implement number 232. Just as AND can implement MAJ by way of A⇛B,C in Figure 3, similarly, 154 can implement 232 by way of 174 in Figure 4, and 174 can be thought of as $\bar{A} \Rrightarrow \bar{B}, C$.

4 Proof Methods for Finding the Lattice

The previous section presented results in the form of lattice diagrams, and these diagrams are a compact representation of the output of many dozens of proofs. In this extended abstract, we will simply indicate the flavor of the proof styles, and try to expose the main ideas of some of the larger proofs. We can assert with confidence that the proofs themselves range from the level of easy homework problems, to hard homework problems, to questions that can be puzzled over for months.

Every connection or lack of connection in the lattice is a theorem that must be proved. The connections are easy to prove: Just show the implementation. An example is shown in Figure 5. Each lack of connection generally requires giving some characterization, however slight, of the sorts of predicates that are implementable by a predicate X, and then showing that Y does not fit this characterization. As a simple example, any graph of self-dual predicates (predicates where negating every variable has no effect on acceptance) can clearly only implement another self-dual predicate (even using free negation, since negation is also self-dual), so "$\neq_3$", a self-dual predicate, cannot implement "0" (the predicate that forces every variable to be zero). As a consequence, we know that "$\neq_3$" cannot implement any of the predicates that are above "0" in the lattice, for if it could, then it would be able to implement "0" as well, due to the transitivity of implementability.

An interesting fact is that AND, which can be seen in Figure 3 to be by far the most capable predicate, cannot implement "$=_3$" (fan-out). This can be proven by induction on the number of vertices in the graph which do not have self-loops: Assuming you are given an implementation of fan-out, you can show by a case analysis of the graph structure near a dangling edge that any implementation can be reduced to a smaller one.

Many such proofs are required, of varying difficulty. The hardest part of Figure 3 to prove, for us, was that AND cannot implement XOR. If the reader can find a simple proof of this fact, we would be very interested to hear about it. Our proof involves considering minimally different witness assignments for three of the triples accepted by XOR, and then using the structure of the differences within the graph to show that another triple must also be accepted by the graph, even though it should be rejected by XOR.

In the area of computational complexity, proving that only NAND can create NP-hard satisfiability problems only requires proving statements about the three peaks and the points directly under the third peak in each lattice, since any lower predicate in the lattice can have its graphs converted to graphs of one of these higher predicates with only a constant factor of growth in graph size. These complexity results are in general easier than finding the structure of the lattice itself.

5 Undecidability in the Lattice

The variety and difficulty of the proofs involved in finding the lattice led us to wonder whether there might be some reason for the difficulty. Eventually we were able to answer this in the affirmative, by proving the undecidability of the general problem.

Most of the undecidability results rest on the following main theorem:

Theorem 1 *There is no general procedure for deciding the following question: Given a predicate X, can a graph of X's be built that implements a predicate that rejects a given tuple T?*

The proof of this theorem is long and involved, and is described in the Appendix.

From this theorem, many corollaries immediately follow, showing that many questions of implementability are undecidable:

Theorem 2 *There is no general procedure for deciding the following question: Given a predicate X, can a graph of X's be built that implements a given desired target predicate Y?*

Proof: If there were a procedure for deciding this question, we could use it to decide whether or not X can implement Y for every possible Y relating a given number k of values (there are only a finite number of such Y, since the values must be from the finite alphabet of values that can be accepted by a variable participating in X). This would then tell us whether or not any particular k-tuple T can be rejected by a graph of X's, which goes against our main theorem. ■

We can also consider implementability questions involving *sets* of predicates rather than just a single predicate:

Theorem 3 *There is no general procedure for deciding the following question: Given a* set *of predicates $\mathcal{X}$, can a graph of predicates from $\mathcal{X}$ be built that implements a predicate that rejects a given tuple T?*

Proof: Trivial: Consider a singleton set $\mathcal{X}$ and use Theorem 1. ■

Theorem 4 *There is no general procedure for deciding the following question: Given a set of predicates $\mathcal{X}$, can a graph of predicates from $\mathcal{X}$ be built that implements a given desired target predicate Y?*

This can be proved just like the previous ones. In some sense this seems like the most natural or general way to phrase the question of implementability. If the question *were* decidable, this is what you would want the decision procedure to be able to do. In the next section, we will see that if the "fan-out" predicate (the equality predicate on three variables) is in the set $\mathcal{X}$, then this question becomes decidable!

Theorem 5 *There exists a fixed predicate X for which there is no general procedure for deciding the following question: Can a graph of X's reject a given set of tuples $\mathcal{T}$?*

This is not really a corollary, as it does not follow from the main theorem. However, it can be proved along almost identical lines as the main theorem, as described at the end of the Appendix. Analogous corollaries follow from this theorem as well.

The proof of our main theorem constructs an X with a large alphabet and a small number of variables (three variables). Is the question still undecidable if X has a small alphabet (but more variables)? The answer turns out to be yes (Boolean values are sufficient), but new ideas are needed for the construction in this case. Of course, if one limits X to both a small (bounded) alphabet and a small (bounded) number of variables, then there are only a finite number of possible distinct choices for X, and so all such questions about small predicates are necessarily decidable. (For example, all such questions about predicates on just three Boolean variables can be decided by referencing Figure 4.)

If infinite graphs are allowed (which strikes many mathematicians as natural, but not too many other people), then the undecidability results are unchanged, although the proofs must be modified.

6 Decidability in the Lattice

The following theorem highlights the different nature of acceptance and rejection.

Theorem 6 *There is a general procedure for deciding the following question: Given a set of predicates $\mathcal{X}$, can a graph of predicates from $\mathcal{X}$ be built that implements a predicate that* accepts *a given tuple T?*

Proof: The idea here is that each acceptable tuple in each predicate in $\mathcal{X}$ contains either an even or an odd number of instances of each possible variable value. We can think of each of these acceptable tuples as a 0/1 vector of length s, if there are s values appearing among the acceptable tuples. The given tuple T also contains either an even or an odd number of instances of each variable value. If we can find a set of 0/1 vectors whose sum (mod 2) matches the tuple T (which is a straightforward linear algebra problem), then we just lay them all out on the table and start connecting identical values in pairs, and the end result is the desired graph. If we cannot find such a set, then clearly no accepting graph can be built. (To avoid connecting a pair of values appearing in T, we may need to add a pair of identical acceptable tuples containing that value (assuming such a tuple exists), which then allows us to connect values in pairs so that the pair of instances of that value in T are connected to predicates in the graph rather than directly to each other.) ■

Even asking about the acceptance of entire *sets* of tuples leaves us in the decidable world:

Theorem 7 *There is a general procedure for deciding the following question: Given a set of predicates $\mathcal{X}$, can a graph of predicates from $\mathcal{X}$ be built that implements a predicate that accepts a given* set *of tuples $\mathcal{T}$?*

Proof: This problem can be reduced to the previous problem by replacing the set of variable values $\mathcal{S}$ with a larger set of variable values. If there are m tuples in the set of tuples $\mathcal{T}$, then we collapse $\mathcal{T}$ into a single tuple T, each of whose values is a member of the set $\mathcal{S}^m$. The available predicates in $\mathcal{X}$ can be similarly converted into predicates on the larger alphabet $\mathcal{S}^m$. We can then continue as in the proof of the previous theorem, using this larger alphabet of values. ■

It is at first surprising that the decision problem for acceptance should be so easy when the decision problem for rejection is undecidable. One way to understand this intuitively is that a graph accepts a tuple if "$\exists$ an assignment of values to edges such that $\forall$ predicates in the graph, the predicate is satisfied." On the other hand, a graph rejects a tuple if "$\forall$ assignments of values to edges, $\exists$ a predicate that is not satisfied." Now, we have been considering questions of the form "Does there exist a graph that accepts/rejects something?" So for acceptance, we are prepending an existential quantifier to something that already started with an existential quantifier, whereas for rejection, we are prepending an existential quantifier to something that started with a universal quantifier, and the additional level of alternation manages to complicate the problem considerably.

Our final example of a decidable theorem of this nature is the following theorem about sets of predicates that include fan-out (the equality predicate on three variables). It says that if we ask the same implementability question that was undecidable before, now the presence of fan-out will make it be decidable. Much previous work exists on the analysis of predicates where fan-out is available [1, 2, 6, 9]. Here we present a proof due to Pippenger [9], followed by a significant improvement in the algorithmic complexity of the decision procedure.

Theorem 8 *There is a general procedure for deciding the following question: Given a set of predicates $\mathcal{X}$ that includes an equality predicate on three or more variables (or that can implement such a predicate), can a graph of predicates from $\mathcal{X}$ be built that implements a given desired target predicate Y?*

Proof: The main idea is the following: Suppose a graph accepts and rejects certain sets of tuples, $\mathcal{T}_{\text{acc}}$ and $\mathcal{T}_{\text{rej}}$. For each accepted tuple, there is by definition some assignment of values to edges that is acceptable to every predicate. We will arbitrarily choose one particular such accepting assignment for each accepted tuple and call it the witness assignment for that tuple. For any set of two or more edges that have the same value in every witness assignment, we will replace those edges with a connected graph of equality predicates, effectively connecting those edges together so they are forced to always have the same value. This new graph will still be able to use effectively the same witness assignments to accept every tuple that was previously accepted, and the new restrictions (forcing certain edges to always match in value) will certainly not allow any previously rejected tuples to become acceptable, so the new graph is implementing the same predicate that the old graph implemented. If we use just a single variable for each subgraph of edges connected by equality predicates (as if it were a hyperedge for a hypergraph version of the problem), and there are s possible values that a variable can have, then we see that there are at most $s^{|\mathcal{T}_{\text{acc}}|}$ variables, since any edges matching in value on each of the $|\mathcal{T}_{\text{acc}}|$ witnesses have been connected to use a single variable. Since there is a bound on the number of variables, this means there is a bound on the number of ways a predicate can be applied to the variables, and thus a bound on the number of possible graphs, and thus the problem is decidable by exhaustive search. ■

To Pippenger's proof we add the comment that actually the search can be narrowed down to simply checking one single graph which will implement the target predicate if anything can. This graph has all $s^{|\mathcal{T}_{\text{acc}}|}$ variables, and relates them in all possible ways (compatible with $\mathcal{T}_{\text{acc}}$) using predicates from $\mathcal{X}$. The predicates are therefore compatible with the values of the variables for each of the $|\mathcal{T}_{\text{acc}}|$ required solution states of the graph, but the graph is maximally restrictive subject to accepting what it needs to. All that then needs to be checked is whether the target predicate Y is indeed enforced on the set of variables to which it would be connected if it were in $\mathcal{X}$.

This theorem shows how the presence of fan-out can have a surprising effect on our ability to analyze the power of a given set of predicates.

Indeed, when fan-out is present, not only does implementability become decidable, but there are some powerful theorems ([1, 6]) that can convert any question of implementability for predicates into an implementability question for functions (which combine to implement other functions using regular function composition), and vice versa, by means of a *Galois connection* between the two. The implementability question for functions has been studied since Post [10], who in fact investigated a more general model than just what the Galois connection applies to. Despite trying, we have not been able to generalize the Galois connection so as to apply to general predicate graphs.

7 Open Questions

Our work so far has focused on whether certain predicates can implement certain others, and on the decidability of that question. In this paper we have ignored the issue of how large a graph may be required for an implementation (although our undecidability proof implies that in the most general case it may need to be uncomputably large), or how many times a particular predicate must be used. Many such questions can be imagined. Of course, finding lower bounds for the size of a predicate graph implementation can directly yield equivalent bounds for traditional circuit complexity when using the characteristic relations of the corresponding gates. In light of this, it is likely that such bounds will be difficult to obtain, although very valuable if found.

Here is a more concrete question:

Open Question 1 *Is there a general procedure for deciding the following question: Given a predicate X, can a graph of X's be built that is not satisfiable?*

This question asks about graphs with no dangling edges. In terms of implementability, the target predicate to be implemented in this case is the trivial empty predicate on no variables. This question seems similar in spirit to the challenge of removing the "seed tile" from the constructions that show that the tiling problem is undecidable. The dangling edge in our construction seems analogous to the seed tile in tiling constructions—it gives us an anchor point from where we can start building our construction. Perhaps, as was the case with tilings, completely new ideas will be needed to show that this question is undecidable too.

Acknowledgments

We would like to thank Erik Winfree for helpful discussions. This research was supported in part by the "Alpha Project" that is funded by a grant from the National Human Genome Research Institute (Grant No. P50 HG02370).

References

[1] V. G. Bodnarchuk, L. A. Kaluzhnin, V. N. Kotov, and B. A. Romov. Galois theory for Post algebras. *Kibernetika*, 5(3):1–10, May-June 1969.

[2] A. A. Bulatov, A. Krokhin, and P. Jeavons. The complexity of maximal constraint languages. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 2003.

[3] M. Cook and J. Bruck. Networks of relations for representation, learning, and generalization. In A. Abraham and J. Abonyi, editors, *Proceedings of the Fourth International Conference on Intelligent System Design and Applications*, Advances in Soft Computing. Springer-Verlag, 2004.

[4] R. Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[5] T. Feder. Fanout limitations on constraint systems. *Theoretical Computer Science*, 255(1–2):281–293, March 2001.

[6] D. Geiger. Closed systems of functions and predicates. *Pacific Journal of Mathematics*, 27:95–100, 1968.

[7] J. Lagarias. The $3x+1$ problem and its generalizations. *American Mathematical Monthly*, 92:3–23, 1985.

[8] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice Hall, 1967.

[9] N. Pippenger. *Theories of Computability.* Cambridge University Press, 1997.

[10] E. L. Post. *On The Two-Valued Iterative Systems of Mathematical Logic.* Princeton University Press, 1941.

[11] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 216–226. Association for Computing Machinery, 1978.

[12] H. Wang. Proving theorems by pattern recognition II. *Bell System Technical Journal*, 40:1–42, 1961.

APPENDIX

The Big Undecidability Proof

This appendix describes the proof of Theorem 1.

The way we prove Theorem 1 is by showing that for any program P, we can set T to be a simple one-tuple (just a single value to reject on a single dangling edge), and then we can carefully construct a predicate X on three variables such that the only way a graph of X's can reject T is for the graph to have a structure that corresponds to the execution of P, with P halting. So if P halts, then it is possible to build a (unique) graph of X's that rejects T, but if P does not halt, then any graph of X's will accept T. Since this reduces the halting problem to a question of the form given in Theorem 1, it proves the theorem.

Our construction is based on programs P that are "register machine" programs (also known as "counter machine" or "Minsky machine" programs—Minsky himself called them "program machines." [8]) They work like a simplified assembly code, in which the only operations are incrementing and decrementing one of a fixed number of registers, and the decrement operation can branch according to whether the register was zero (undecrementable) or not.[2] An example will be given below.

In constructing the predicate X (based on P), we find ourselves doing a very strange kind of programming. Each part of the strange program encoded in the acceptable tuples of X is essentially a detector which checks to see if the graph topology does not correspond in the intended way to the execution of program P. That is, the strange program in X is built up of pieces that specify what should *not* happen when program P runs. Each piece is designed so that if the undesired topology can be detected, then the one-tuple T will be acceptable to the graph. The strange program in X is complete when all undesired topologies have been excluded, and the only remaining possibility for the graph topology is to be a perfect representation of the execution of P. The pieces of the strange program in X only get "run" if the graph topology is failing to represent the execution of P in the way being checked for by that piece of the strange program. If the graph topology corresponds perfectly to the execution of P, then no part of the strange program in X can be run, and thus T is not accepted by the graph.

Thus, the construction here has a rather different flavor from most undecidability constructions, in which one shows how some new and different simple system is capable of performing computation. Here, we take a new and different simple system, and show how we can get it to detect *all* the situations where it is *not* performing computation. Like sculpting with chisel and stone instead of with wood and glue, instead of making gadgets that slowly build up more calculational power until the desired program can be executed, this construction has gadgets that chip away at miscalculations until all that is left is proper execution of the desired program.[3]

[2]Different texts often use slightly different definitions, but these differences are never of consequence for the results.

[3]This aspect of this proof is similar to the standard proof that it is undecidable whether a context-free grammar generates all strings, although that proof is far more straightforward.

Register Machines

As an example of a register machine program in the format we will use, we present one that implements the well-known "$3x+1$" procedure [7]. This procedure manipulates a positive integer value repeatedly in the following way: If the number is even, divide it by two, but if it is odd, then multiply it by three and add one. This process is repeated until the value is eventually reduced down to 1. The well-known conjecture is that the value 1 is indeed eventually reached no matter what positive integer value the procedure starts with.

The program below uses two registers, x and y, and is written in the format "line number, increment, next instruction" for the incrementing instructions, and "line number, decrement, next instruction if decrement succeeded, next instruction if decrement failed (because register is already 0)" for the decrementing instructions.

	action	next	but if zero
1.	x-	2	4
2.	y+	3	
3.	x-	1	8
4.	y-	5	6
5.	x+	4	
6.	x-	7	5
7.	x-	9	halt
8.	y-	9	3
9.	x+	10	
10.	x+	11	
11.	x+	8	

As an example of why it is hard to show that this program always halts, note that if it is started with the value 9 in both x and y, it will toil away for over $100,000$ steps before halting!

The "$3x+1$" conjecture corresponds to the conjecture that this machine will always eventually halt, regardless of the initial values of its registers.

For our proof, we will need to use programs with a few specific properties. The register machine programs that we use in our proof will need to be programs that start with 0 in all the registers. If a program needs to start with a nonzero value in some register, it can simply increment the register to the desired value at the beginning of the program. Any "input" to the program is thus treated as part of the program itself. Another feature of the programs used in the proof will be that they should have 0 in all the registers whenever they halt. If a program might halt with nonzero registers, we can simply attach loops to the end of the program to decrement all the registers back down to zero before halting. We will also require the program instructions never to point to themselves as the next instruction. If a program needs to execute a single line repeatedly, we can simply duplicate the line and let the execution go back and forth between the two lines. Thus, although our proof will only address the halting problem for programs of a certain form, we see that any program can easily be converted to the required form.

The Graph Corresponding to P's Execution

Here we simply define what the graph corresponding to P's execution should look like.

The graph will be built using many copies of a single predicate on three variables, and the graph will have a single dangling edge. (This edge is where it will reject a value.)

The building-block predicate, X, will be a predicate on three values, which we will call A, B, and C. So three edge-ends will meet at each vertex in the graph. The predicate at each vertex knows which edge is which—it knows which edge's value will be treated as A, which edge's value will be treated as B, and which edge's value will be treated as C. Of course, it may be the case that at one end, an edge's value will be treated as the A value in one predicate, while at the other end, it will be treated as the C value in another predicate. It may even be the case that an edge loops from a vertex back to the same vertex, so its value might be used as both the A value and the C value of a single particular predicate. Such edges are called *self-loops*. (If for some reason self-loops are unacceptable, parts of the proof become much more complicated, but the nature of the proof does not change.)

Where an edge is treated by a vertex as the A (or B, or C) value for that predicate, we will say that that edge is the *A-connection* (or *B-connection*, or *C-connection*) for that predicate.

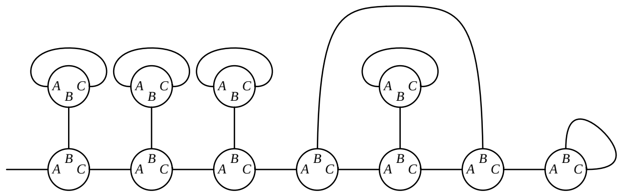
The main feature of the graph corresponding to P's execution will be a backbone consisting of an A-C chain of predicates. What we mean by this is that the dangling edge will be the A-connection of the first predicate in the backbone, and that predicate's C-connection will then be the A-connection of the next predicate in the backbone, and so on, until finally the backbone ends with a self-loop from the C-connection back to the B-connection of the final predicate.

Each predicate along the backbone will correspond to one step in the execution of P. All that remains to be specified is the B-connections of all the predicates in the backbone (except the final one). We refer to these B-connections as the *hairs* on the backbone. The hairs will represent the values of the various registers.

Since all the registers start and end at zero, it must be possible to pair up the increment instructions with the decrement instructions for any given register, so that each increment instruction is paired with a later decrement instruction for the same register. (The decrement instructions which fail due to the register already being 0 will not participate in this pairing.) In the graph corresponding to P's execution, this pairing of instructions is represented by simply connecting their hairs. So if the 100th instruction increments register y and is paired with the 200th instruction (which decrements y), then an edge will connect the 100th predicate in the backbone to the 200th predicate in the backbone, and it will be the B-connection of both of them.

The only thing left to specify is what happens with the hairs for decrement instructions that fail because the register is already 0. For each such instruction, we will add a new vertex that has an A-C self-loop, and we will connect the instruction's vertex (which is in the backbone) to this new vertex (which is not in the backbone) with an edge that is the B-connection of both predicates.

This completes our specification of the graph corresponding to the execution of P. As a very simple example, here is the graph corresponding to the execution of the register machine program shown above, with registers started at zero.



The execution of that program, with the registers started at zero, progresses through the following sequence of lines: $(1, 4, 6, 5, 4, 6, 7, halt)$. Five of the seven executed instructions are decrement instructions that fail because the registers are already 0. These cases are easily spotted in the graph structure due to the "parking meter" predicate planted atop each one (named for its visual appearance in this diagram).

If we imagine that all the hairs of backbone predicates for instructions manipulating the first register are colored green, then we see that the number of green hairs passing over an edge of the backbone gives the value that is in the first register at that stage of running the program. In particular, a green parking meter post cannot occur under a green increment/decrement pair hair, because that would mean that a decrement failed when the register was not in fact 0, which does not happen.

Now that we have a reasonable understanding of the graph corresponding to P's execution, let us look at exactly what predicate on A, B, and C we can use so that the requirement (of rejecting a particular value at the dangling edge at the start of the backbone) will force the graph to be the one corresponding to P's execution.

The Construction of the predicate X

The easiest way to explain X's construction is just to give it, and then show why it works. To really understand it, you will probably have to get out your pencil and convince yourself of why it does what we say it does, as we walk through it. Here it is:

A	B	C
e_1	e_1	$\#_1$
e_1	$\#_1$	e_1
e_1	e_1	e_1
$\#_1$	e_2	P_2
P_2	e_2	P_2
P_2	e_2	Q_2
e_2	Q_2	e_2
e_2	e_2	Q_2
P_2	Q_2	P_2
$\#_1$	Q_2	P_2
$\#_1$	e_2	Q_2
e_2	e_2	e_2
$\#_1$	e_3	R_3
R_3	e_3	R_3
R_3	S_3	T_3
$\#_1$	S_3	T_3
T_3	e_3	T_3
T_3	U_3	U_3
e_3	e_3	e_3
S_3	e_3	e_3
e_3	e_3	S_3
e_3	S_3	V_3
e_3	e_3	V_3
e_3	V_3	e_3
V_3	e_3	e_3

A	B	C
$\bowtie$	$\triangledown$	$\bowtie$
$\bowtie$	$\blacktriangledown$	$\bowtie$
$\#_{k:inc}$	$\frown$	$\#_{k.next}$
$\#_{k:dec}$	$\frown$	$\#_{k.next_{\neq 0}}$
$\#_{k:dec}$	$\triangledown$	$\#_{k.next_{=0}}$
$\#^i_{k:inc}$	$\frown$	$\#^i_{k.next}$
$\#^i_{k:dec}$	$\frown$	$\#^i_{k.next_{\neq 0}}$
$\#^i_{k:dec}$	$\triangledown$	$\#^i_{k.next_{=0}}$
$\square$	$\frown$	$\boxdot$
$\boxdot$	$\frown$	$\square$
$\square$	$\triangledown$	$\boxdot$
$\boxdot$	$\triangledown$	$\square$
$\square$	!	$\bigcirc$
$\boxdot$	!	$\bigcirc$
$\bigcirc$	$\frown$	$\odot$
$\odot$	$\frown$	$\bigcirc$
$\bigcirc$	$\triangledown$	$\odot$
$\odot$	$\triangledown$	$\bigcirc$
$\bigcirc$	$\circlearrowright$	$\circlearrowright$
$\odot$	$\circlearrowright$	$\circlearrowright$
$\#_{k:inc}$	$\blacktriangledown$	$\bigcirc$
$\#_{k:inc/dec}$	$\circlearrowright$	$\circlearrowright$
$\#_{k:halt}$	$\frown$	$\bigcirc$
$\#_{k:halt}$	$\triangledown$	$\bigcirc$
$\#_{k:dec}$	!	$\square$
$\#_{k:inc\ r_i}$	!	$\#^i_{k.next}$
$\#^i_{k:inc}$	!	$\bigcirc$
$\#^i_{k:dec\ r_{j\neq i}}$	!	$\bigcirc$
$\#^i_{k:dec\ r_{j=i}}$	$\blacktriangledown$	$\square$
$\#_{k:inc\ r_i}$	!!	$\#^i_{k.next}$
$\#^i_{k:inc\ r_{i=i}}$	!	$\pentagon$
$\pentagon$	$\frown$	$\dot\pentagon$
$\dot\pentagon$	$\frown$	$\pentagon$
$\pentagon$	$\triangledown$	$\dot\pentagon$
$\dot\pentagon$	$\triangledown$	$\pentagon$
$\pentagon$	!!	$\square$
$\dot\pentagon$	!!	$\square$

The finite alphabet consists of all the symbols appearing in the table:

$$\{e_1, e_2, e_3, P_2, ..., \#_1, \#_1^x, \#_1^y, \#_2, ..., \bowtie, \triangledown, \blacktriangledown, \frown, ...\}$$

Despite the subscripts, these symbols are not variables; they are values that a variable might have. The subscripts merely serve to help organize them. Only i and k (as superscripts and subscripts to the # symbol) need to be substituted for to get actual symbols: i with a register name, and k with a line number of the program.

The one-tuple to be rejected is $\langle \#_1 \rangle$. So the graph must have exactly one dangling edge, and we want to know if there is such a graph that is unsatisfiable when that edge has the value $\#_1$.

The first three triples shown above for predicate X force the edge dangling out of the graph to be the A edge of the predicate it is dangling from. For if it were not the A edge, the first three triples shown above would allow it to have the value $\#_1$, and every other edge in the graph could have the value e_1, and thus $\#_1$ would be acceptable, even though it is supposed to be rejected.

The next nine triples shown above for X force the A-C chain to end with a self-loop from the final C to the B of the same predicate. In any other case, $\#_1$ will be acceptable with the edges in the A-C chain having value P_2, the final edge from the C at the end of the chain (which, being the end of the chain, must go to a non-A connection) having value Q_2, and all other edges in the graph having value e_2.

The next nine triples shown force the B connections along the backbone to either connect to another predicate on the backbone, or to connect to the B connection of a predicate not on the backbone.

The final four triples in the first column work together with the previous nine to place further restrictions when a B on the backbone is connected to a B not on the backbone: In this case, the predicate not on the backbone must have an A-C self-loop, since otherwise its C connection could have the value V_3 while every other edge (including its A connection) has value e_3.

So the first column forces the graph to consist of a backbone with hairs either doubly-connected or leading to parking meters. So the topology looks like *some* program, but we have yet to make sure it is the *right* program.

We now move to the second column of triples.

Here, the first two triples are the only ones, of all the remaining triples, that have the same value for the A and C connections, so one of them must be used on each parking meter in any remaining solution.

The next three triples actually represent many triples, based on the program P. The expression $\#_{k:inc}$ represents a different symbol (namely $\#_k$) for each line k of the program P that is an increment instruction. The "*inc*" is just for our reference in knowing what triples should be created based on this "proto-triple." The expression $\#_{k.next}$ represents the value $\#_j$ for that j which the program indicates is the next instruction to be executed after the increment instruction at line k. The proto-triples for decrement instructions work similarly. All the triples indicated by these three proto-triples work together to allow an initial segment of the backbone to be filled with $\#_k$ values on each edge, with k progressing exactly as the current instruction progresses during the execution of P.

The next three proto-triples are exactly the same, except that they create even more triples: one for each possible value of i, where i identifies one of the registers (but not the register's value). The value of i is independent of the instruction at line k. So if there are two registers, x and y, then these proto-triples would create triples that are superscripted with either the letter x or the letter y, but always the same superscript on both the A connection and the C connection. The purpose of these triples is that they can be used along some stretch of the backbone, thus following the progress of P's execution while "remembering" which of the registers needs to be coordinated between the left and right end of that backbone stretch.

The next six triples allow a square symbol to fill a stretch of the backbone, ending with a single hair with an exclamation mark, at which point a stretch of circle symbols will follow. The dot inside the square has no purpose except to make the A and C values be different for every triple, so as not to interfere with the purpose of the first two triples in the column.

The next six triples allow the stretch of circles to fill the final stretch of the backbone, ending with the C-B self-loop.

Up to here the column has been preparing some infrastructure without forcing any particular structure on the graph, but from here on down, the triples use the infrastructure to actively enforce things. At the first error in the graph representation of P's execution, they will take advantage of the error to allow $\langle\#_1\rangle$ to be accepted.

The next proto-triple, for example, prevents increment instructions from being connected to parking meters. For if one is, then $\#_k$ instructions can be used leading up to that position, and circles can be used afterwards, and the graph will accept $\langle\#_1\rangle$. The filled triangle is used because it cannot mistakenly occur on a non-parking meter hair.

The next three proto-triples prevent the backbone from ending before the program does (the first proto-triple), or from ending after the program does (the second or third proto-triple).

The next proto-triple prevents a decrement hair from coming back down to the backbone after the decrement instruction. In other words, decrement hairs must have been produced *prior* to the decrement instruction, which also implies they cannot have been produced by a previous decrement instruction, but only by a previous increment instruction.

The final four proto-triples force each increment to be matched to a decrement of the same register, and prevent parking meters from existing when the corresponding register is not zero.

Finally, the eight triples and proto-triples in the short final column of triples are optional, and merely enforce the increments and decrements to be matched like matching parentheses. The purpose of this is to detangle the graph's hair into a unique form. If these triples are included, and there is a graph that rejects $\langle\#_1\rangle$, then that graph is unique. If these triples are not included, then the graph will generally not be unique. The proof will work either way.

It is not hard to see that the various groups of triples cannot combine in unforeseen ways to mistakenly accept $\langle\#_1\rangle$ when each individual group would have rejected it. For example, the distinct indices on the symbols in the first table of triples prevent the groups of triples from being able to interact with each other (except for the third and fourth groups, which are designed to work together).

In summary, if the graph is exactly the graph corresponding to P's execution, then $\langle\#_1\rangle$ will be rejected by the graph, but if the graph is anything else, then $\langle\#_1\rangle$ will be accepted.

Proof Variants

Using a fixed predicate

The main construction creates a different predicate X for different programs. If we want to always use a single predicate X, the natural idea is that X should be the predicate given by the above construction for a fixed register machine program P that is itself universal. Then, the initial values of the registers, when the machine is started, can contain the "real" program for which we would like to know whether it halts or not.

However, getting the initial values of the registers into the graph is not trivial. The initial register values need to be incorporated into the graph by using many lengthy tuples in $\mathcal{T}$ instead of just a single one-tuple. The idea behind the many tuples is that they encode all the values, for the dangling backbone and hair edges, that could have been used in the original construction (which would have had a single dangling edge and started with many increment instructions) to detect one of the many possible types of errors. This moves the initial condition from the triples that form X into the tuples of $\mathcal{T}$. So then, using a universal register machine program P completes the job.

The reader will have noticed that we have failed to simplify $\mathcal{T}$ down to a single tuple for this case. It is an open question whether a fixed predicate X can be constructed for which the question "Given a tuple T, does there exist a graph of X's which rejects T?" is undecidable.

Boolean Alphabet

With a Boolean alphabet, if we want to use roughly the same construction idea, we need to have multiple edges between predicates. This leads to a host of new ways that the predicate graph might fail to accurately represent the running of the program. In particular, the multiple edges might not be properly aligned with each other (regarding their orderings at the vertices), or perhaps might not even all go to the same other vertex. Many gadgets are required to deal with these troubles; we will not list them here. The main idea that allows them to succeed is to use low-weight tuples and high-weight tuples for enforcing multiple edge alignment, while using medium-weight tuples to encode the variable values of the previous construction.

Infinite Graphs

The main difference in the proof, when infinite graphs are allowed, is that now the execution of a non-halting program might be accurately modeled by the topology of an infinite graph. However, whereas for a finite graph, having an accurate topology led to rejection of the proscribed tuple T, in an infinite graph, T can be accepted even if the topology has no errors, since all edges can take the values they would take if there were going to be an error farther down the line. Thus, it is still the case that there is a graph that rejects T if and only if the program halts.