Approximate Bit Dependency Analysis to Identify Program Synthesis Problems as Infeasible

Marius Kamp(✉) and Michael Philippsen

Programming Systems Group,
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
marius.kamp@fau.de, michael.philippsen@fau.de

Abstract. Bit-vector-based program synthesis is an important building block of state-of-the-art techniques in computer programming. Some of these techniques do not only rely on a synthesizer's ability to return an appropriate program if it exists but also require a synthesizer to detect if there is no such program at all in the entire search space (i.e., the problem is infeasible), which is a computationally demanding task.
In this paper, we propose an approach to quickly identify some synthesis problems as infeasible. We observe that a specification function encodes dependencies between input and output bits that a correct program must satisfy. To exploit this fact, we present approximate analyses of essential bits and use them in two novel algorithms to check if a synthesis problem is infeasible. Our experiments show that adding our technique to applications of bit vector synthesis can save up to 33% of their time.

1 Introduction

Program synthesis is the construction of a program that satisfies a declarative specification. Its ability to create a program that implements a specification function and consists of some given bit vector operations has recently propelled research in computer programming. For example, program synthesizers craft instruction selection rules in compilers [4], superoptimize code [18], generate code for unusual architectures [17], optimize machine learning kernels [5], or enumerate rewrite rules for SMT solvers [16]. As it is often not a priori known which and how many operations to use for a synthesized program, some applications formulate multiple synthesis problems that differ in the used operations [4] or in the length of the program [18]. Other works search over a collection of synthesis tasks [2] or generate synthesis problems based on a symbolic execution of a program [14]. All these approaches have a common trait: for some of the synthesis problems, there may not be a program that implements the specification *and* consists of the available operations. These problems are called *infeasible*. Infeasible problems tend to be harder than feasible ones of comparable size because synthesizers have to show that there is no such program in the entire solution space. If infeasible problems occur frequently, the performance of an application "critically depends on the performance of unsatisfiable queries" [14] (i.e., infeasible problems). Thus, applications get faster by quickly identifying infeasible problems as such without invoking the costly synthesizer.

We present such an infeasibility checker for bit vector synthesis problems (f, Ops) with a specification function f plus a collection of available operations Ops that bounds the number of times an operation may occur in the constructed program. To illustrate our approach, our running example is the problem of computing the average of two integers x and y by rounding up fractional solutions. The program $(x \vee y) - ((x \oplus y) \gg 1)$ satisfies this specification [21]. When human programmers want to find such a program, they intuitively know that they need to divide by 2 or right shift by 1 to compute the average (suppose for now that the constant belongs to the operation). Alas, program synthesizers lack this intuition. We can, however, find out that the synthesized program indeed needs such an operation by inspecting the dependencies between input and output bits of the specification: The $i + 1$-th input bits of both x and y influence the i-th output bit. To see this, consider $x = 2$, $y = 0$, and the result 1. If we turn off the second bit of x, the result is 0. Hence, the second bit of x influences the first output bit. Since this bit dependency must also exist in the synthesized program, we need an operation that can provide it. Thus, we analyze the dependencies between input and output bits of both the specification function and each of the available operations. Since this analysis is NP-complete, Sec. 3 proposes approximations. Once we know the bit dependencies of both the specification and the operations, we can check if the operations cannot be combined to satisfy the bit dependencies of the specification. In this case, the synthesis problem is infeasible and we do not need to invoke the synthesizer. We derive a checking algorithm in Sec. 4 that uses an abstraction of the concrete bit dependencies called *shapes*.

The collective bit dependencies of an output bit come in two flavors. These correspond to the value the output bit takes if all of the input bits it depends on are set to 0. All output bits of our running example come in the same flavor, but if we modify the specification to compute the average of $\neg x$ and y instead, the highest (most significant) output bit will take the value 1 for $x = 0$ and $y = 0$. Hence, the highest output bit comes in a different flavor than the other output bits. We give a second algorithm in Sec. 4 that detects if some output bit cannot get the right flavor using the given operations. Although both algorithms may miss infeasible problems, they never flag a feasible problem as infeasible. We evaluate our contributions in Sec. 5 and discuss related work in Sec. 6.

2 Fundamentals

A (fixed-width) *bit vector* is a vector of Boolean variables (the *bits*) of constant length (the *width*). Let B_k be the set of all bit vectors of width k. A *bit vector function* is then a function $B_{k_1} \times \cdots \times B_{k_n} \to B_{k_o}$. If all $k_1, \ldots, k_n, k_o = 1$, the function is a *Boolean function*.

Well-known bit vector functions include bitwise operations like conjunction ($\wedge$), inclusive ($\vee$) and exclusive ($\oplus$) disjunction, arithmetic operations ($+$, $-$, $\times$, $\div$, rem), and bit shifts ($\ll$, $\gg$). The specification function, the operations, and the resulting program are also bit vector functions. Without loss of generality, we assume the same bit width for all of them.

A bit vector function with arguments of width $k_1, \ldots, k_n$ may be transformed into a bit vector function with $\sum_{i=1}^{n} k_i$ arguments of width 1; and also into k_o functions of output width 1. Then, every function represents the calculation of one output bit. Thus, every bit vector function f corresponds to a collection of Boolean functions $f_1, \ldots, f_{k_o}$.

$f_{|x_i=c}(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, c, x_{i+1}, ..., x_n)$ is the *restriction* of f to $x_i = c$. An input variable x_i is *essential in* f if there exist constants $c_1, ..., c_n$ such that $f_{|x_i=0}(c_1, ..., c_n) \neq f_{|x_i=1}(c_1, ..., c_n)$. Intuitively, the essential variables are exactly those variables that influence the result of f.

A Boolean function f can be uniquely represented by the *Zhegalkin polynomial* [6] with coefficients $a_K \in \{0, 1\}$:

$$f(x_1, ..., x_n) = \bigoplus_{K \subseteq \{1, .., n\}} a_K \wedge \bigwedge_{i \in K} x_i.$$

For example, $\neg(x_1 \wedge x_2) = 1 \oplus (x_1 \wedge x_2)$ with $a_{\{1\}} = a_{\{2\}} = 0$ and $a_\emptyset = a_{\{1,2\}} = 1$. The *Reed–Muller decomposition* $f(X) = f_{|x_i=0}(X) \oplus \big(x_i \wedge \big(f_{|x_i=0}(X) \oplus f_{|x_i=1}(X)\big)\big)$ for $X = x_1, ..., x_n$ factors out x_i from this polynomial [21].

In this work, (undirected) graphs must not have multiple edges between the same vertices (*parallel edges*), whereas this is permitted for directed graphs. $\mathrm{N}(X)$ denotes the *neighbors* of all vertices in X. A bipartite graph G (with partition $\{A, B\}$) contains a matching of all vertices of A if and only if $|\mathrm{N}(X)| \geq |X|$ for all $X \subseteq A$ (Hall's Marriage Theorem [8]).

3 Approximation of Essential Bits

We use the dependencies between input and output bits to identify synthesis problems as infeasible. If every program using the given operations violates the bit dependencies of the specification function, the synthesis problem is infeasible. We view the operations and the specification function as collections of Boolean functions f. The essential bits of f provide its bit dependencies. As computing all essential bits of an arbitrary f is NP-complete [6], we approximate them. To do this, we present two underapproximations and one overapproximation.

In this section, we view f as a circuit of $\oplus$ and $\wedge$ gates and the constant 1. Since the well-known bit vector functions have circuits of polynomial size [22], their analysis is still tractable. As both the specification function and custom operations are usually defined with these functions, we can obtain their circuits by combining the circuits of the well-known bit vector functions. A circuit that computes the 2nd bit of our running example is $x_3 \oplus y_3 \oplus (x_1 \wedge (x_2 \oplus y_2) \wedge (y_1 \oplus 1)) \oplus (x_2 \wedge (y_1 \oplus y_2))$ provided that x and y have at least 3 bits. To define our approximations, it hence suffices to supply rules for the constant 1, a bit x_i as well as the operations $\oplus$ and $\wedge$. The cornerstone of our approximations is:

Lemma 1. *The bit x_i is essential in a Boolean function f if and only if the Zhegalkin polynomial of f has at least one coefficient $a_K = 1$ with $i \in K$.*

Proof. If x_i is essential, the Zhegalkin polynomial must contain x_i. Conversely, if x_i is not essential, then $f_{|x_i=0} \oplus f_{|x_i=1} = 0$. By the Reed–Muller decomposition, $f = f_{|x_i=0} \oplus \big(x_i \wedge (f_{|x_i=0} \oplus f_{|x_i=1})\big) = f_{|x_i=0}$. Since the Zhegalkin polynomial of $f_{|x_i=0}$ does not contain x_i, the same holds for f. □

As the number of a_K is exponential in the number of input bits, we do not enumerate all non-zero a_K but only check if some a_K are non-zero. Our underapproximation UA_1 of the essential bits of f only considers the coefficients a_K with $|K| \leq 1$. UA_1 is a set over the x_i and 1. If $x_i \in UA_1$, $a_{\{i\}} = 1$ and x_i is essential according to Lemma 1. The rules to compute UA_1 for 1 and a bit x_i are $UA_1(1) = \{1\}$ and $UA_1(x_i) = \{x_i\}$.

The rule for $f = g \oplus h$ is $UA_1(g \oplus h) = UA_1(g) \triangle UA_1(h)$ with the symmetric set difference $X \triangle Y = (X \setminus Y) \cup (Y \setminus X)$ because the Zhegalkin polynomial of $g \oplus h$ has a non-zero coefficient for $\{i\}$ if either g or h has such a coefficient (since $x_i \oplus x_i = 0$). Thus, $x_i \in UA_1(f)$ iff $x_i \in UA_1(g)$ "xor" $x_i \in UA_1(h)$.

To derive a rule for $f = g \wedge h$, we look at the single monomials g_i of $g = g_1 \oplus \cdots \oplus g_m$ and h_j of $h = h_1 \oplus \cdots \oplus h_n$. Then $f = g \wedge h = (g_1 \oplus \cdots \oplus g_m) \wedge (h_1 \oplus \cdots \oplus h_n) = (g_1 \wedge h_1) \oplus \cdots \oplus (g_1 \wedge h_n) \oplus \cdots \oplus (g_m \wedge h_1) \oplus \cdots \oplus (g_m \wedge h_n)$. For two monomials g_i and h_j, $g_i \wedge h_j$ forms again a monomial and hence has a single non-zero coefficient. Let a_G and a_H be the non-zero coefficients of g_i and h_j respectively. Since the conjunction is idempotent (i.e., $x_i \wedge x_i = x_i$), the non-zero coefficient of $g_i \wedge h_j$ is $a_{G \cup H}$.

Using the known rules, we obtain $UA_1(g \wedge h) = UA_1(g_1 \wedge h_1) \triangle \cdots \triangle UA_1(g_1 \wedge h_n) \triangle \cdots \triangle UA_1(g_m \wedge h_1) \triangle \cdots \triangle UA_1(g_m \wedge h_n)$. Since UA_1 only considers coefficients a_K with $|K| \leq 1$, we can ignore all those $g_i \wedge h_j$ that do not have such a coefficient. For $|G \cup H| \leq 1$, either $G = H$ and $|G| = |H| \leq 1$ or $|G| = 1$ and $H = \emptyset$ (or vice versa). We thus group the $UA_1(g_i \wedge h_j)$ into three sets.

By definition of UA_1, all those monomials that fulfill the first condition are included in $UA_1(g) \cap UA_1(h)$. The set of monomials satisfying the second condition depends on the presence of the NeuTral element 1:

$$\mathrm{NT}(k_1, k_2) = \begin{cases} UA_1(k_1) \setminus \{1\} & \text{if } 1 \in UA_1(k_2) \\ \emptyset & \text{otherwise.} \end{cases}$$

As either G or H may be empty according to the second condition, $UA_1(f)$ depends on $\mathrm{NT}(g, h)$ and $\mathrm{NT}(h, g)$. Since these sets and $UA_1(g) \cap UA_1(g)$ are not necessarily disjoint, the above expansion of $g \wedge h$ requires that we take their symmetric difference: $UA_1(g \wedge h) = (UA_1(g) \cap UA_1(h)) \triangle \mathrm{NT}(g, h) \triangle \mathrm{NT}(h, g)$. For the 2nd bit of our running example, UA_1 holds $\{x_3, y_3\}$.

In general, a combination of $UA_1(g)$ and $UA_1(h)$ does not yield an underapproximation of $f = g \circ h$. For example, let $g(x_1, x_2) = x_2 \oplus (x_1 \wedge x_2)$ and $h(x_1, x_2) = x_1 \oplus x_2 \oplus (x_1 \wedge x_2)$. Then, $x_2 \in UA_1(g)$ but $x_2 \notin UA_1(g \circ h)$ since $g(h(x_1, x_2), x_2) = x_2 \oplus (x_1 \wedge x_2) \oplus x_2 \oplus (x_1 \wedge x_2) = 0$. Thus, to compute $UA_1(g \circ h)$, the circuit for $g \circ h$ must be constructed explicitly.

Next, we show how to use UA_1 to compute the set UA_2 of the coefficients a_K with $|K| \leq 2$. For some bit x_i, the Reed–Muller decomposition splits a function

f into a part $f_{|x_i=0}$ that does not depend on x_i and a part $x_i \wedge (f_{|x_i=0} \oplus f_{|x_i=1})$ in which every monomial contains x_i. Hence, $UA_1(f_{|x_i=0} \oplus f_{|x_i=1})$ contains all those x_j such that $a_{\{i,j\}} = 1$ in the Zhegalkin polynomial of f. If this set is non-empty, x_i and all $x_j \in UA_1(f_{|x_i=0} \oplus f_{|x_i=1})$ are essential bits of f. By taking the union of these essential bits for all bits x_i of f (and additionally $UA_1(f)$), we obtain $UA_2(f)$. For the 2^{nd} bit of our running example, UA_2 is $\{x_1, x_2, x_3, y_1, y_2, y_3\}$ and thus contains all essential bits.

In contrast, our overapproximation OA holds those x_i that possibly occur in some set K with $a_K = 1$ as well as 1 if $a_\emptyset = 1$. As these x_i may occur in arbitrary monomials, we have to adjust our rules. (The rule whether $1 \in OA(f)$ is a special case. It is the same as for UA_1 since this rule is exact in this case.) Again, the first two are $OA(1) = \{1\}$ and $OA(x_i) = \{x_i\}$.

$f = g \oplus h$ has exactly those non-zero coefficients that occur in either g or h. The number of non-zero a_K may be exponential. But since every non-zero coefficient of f is included in the set union of the non-zero coefficients of g and h, the set union overapproximates them: $OA(g \oplus h) = OA(g) \cup OA(h)$.

If x_i is neither essential in g nor h, it is also not essential in $f = g \wedge h$. We can thus also resort to a set union but with one caveat. If g or h is the constant 0 function, the result should also be the constant 0 function, which has no essential bits. Note that $OA(g) = \emptyset$ implies $g = 0$ (similar for h):

$$OA(g \wedge h) = \begin{cases} \emptyset & \text{if } OA(g) = \emptyset \text{ or } OA(h) = \emptyset \\ OA(g) \cup OA(h) & \text{otherwise.} \end{cases}$$

For the 2^{nd} bit of the running example, OA is the same as UA_2 and hence also yields an exact result. In contrast to UA (both UA_1 and UA_2), replacing a bit in $OA(g)$ by $OA(h)$ (i.e., removing x_i and adding $OA(h)$ instead) overapproximates the essential bits of $g \circ h$. As the UA and OA of bit vector functions is the collection of the UAs and OAs of their corresponding Boolean functions, we can derive the OA of a program from the OA of its operations.

4 Flagging Synthesis Problems as Infeasible

The specification function f and its implementing program P represent equivalent bit vector functions and have the same essential bits for each output bit. In terms of approximations, the following two tests $UA(f) \subseteq OA(P)$ and $UA(P) \subseteq OA(f)$ must hold. Hence, to check whether a synthesis problem is infeasible, it suffices to either show that the allowed operations do not admit a program P_U (the *upper bound*) with $UA(f) \subseteq OA(P_U)$ or a program P_L (the *lower bound*) with $UA(P_L) \subseteq OA(f)$. Our definition does not require that upper and lower bounds perform the same computation as P. Proving that these do not exist for a function f is as costly as identifying infeasibility with a standard synthesizer. As our approximations require to build the full circuit for P_L before evaluating $UA(P_L)$, we do not attempt to prove the non-existence of lower bounds. Instead, we show that there is no upper bound by tackling the

complementary problem: If we cannot find an upper bound for f using the approximations of essential bits, the synthesis problem must be infeasible. But it is not necessary to examine all possible programs during this search: With relaxed upper bounds (Sec. 4.1) we only need to consider programs that are trees (Sec. 4.2) and are constructed by expanding a program consisting of at most two operations (Sec. 4.4); and we can use bipartite matchings (Sec. 4.3) to speed up the search. We give an algorithm relying on essential bits (Sec. 4.4) and another one that determines the "flavor" of the bit dependencies by tracking the 1 in the Zhegalkin polynomial (Sec. 4.5).

Note that loop-free programs correspond to *directed acyclic graphs* (DAG) with a single source node (the result). Its nodes are input variables and operations, its edges point to the operands of operation nodes. We call an outgoing edge of an operation o an *argument edge* (short: arg-edge) of o.

4.1 Bit Dependency Shapes

We discovered that the essential bits of an input variable often follow one of four regular patterns that we call *bit shapes*. When the essential bits follow a *simple* shape (symbol: $\diagup$), the i-th output bit is solely based on the i-th input bit, e.g., in operations like $\wedge$. In an *ascending* shape ($\blacktriangle$), input bits $j \leq i$ influence the i-th output bit, e.g., in $+$. In a *descending* shape ($\blacktriangledown$), the i-th output bit depends on input bits $j \geq i$, e.g., the bits of x in $x \gg y$. In a *block* shape ($\blacksquare$), arbitrary input bits influence an output bit, e.g., the bits of y in $x \div y$.

These shapes are partially ordered and form a lattice [7]: a larger shape subsumes all input-output dependencies of a smaller shape. Fig. 1 shows this ordering. $\blacktriangle$ and $\blacktriangledown$ are incomparable, denoted by $\blacktriangle \parallel \blacktriangledown$. The smallest/largest element is $\diagup$/$\blacksquare$.

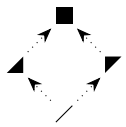


Fig. 1. Lattice.

The shape of an input variable of a specification function resp. the shape of an operand of an operation is the smallest shape that fits to the approximated essential bits of the input variable or operand. We discard both the operands with no influence on the output (no shape) and the operations without operands (e.g., constants). For example, the program that computes the average from Sec. 1 uses the operations $\wedge$, $\oplus$, $\gg 1$, and $-$. For two input variables x and y, the i-th output bit of $x \wedge y$ depends only on the i-th input bit of both x and y. Since this pattern corresponds to a $\diagup$ shape, we say that $\wedge$ is an operation with two inputs in shape $\diagup$. Hence, the list $[\diagup, \diagup]$ is an abstraction of $\wedge$. Similarly, $[\diagup, \diagup]$, $[\blacktriangle, \blacktriangle]$, and $[\blacktriangledown]$ are the shape abstractions of $\oplus$, $-$, and $\gg 1$ respectively.

We derive the shapes of the input variables of a program P via its DAG representation and the computed shape abstractions of its operations. Since a shape abstraction contains one shape per operand, each arg-edge of the DAG is associated with a shape. According to Sec. 3, for an operation o_1 that uses the result of its operand o_2 we can obtain an overapproximation of $o_1 \circ o_2$ by replacing an essential bit x_i in $OA(o_1)$ by all elements of OA of the i-th Boolean function of o_2. To transfer this property to shapes, consider for example $x - (y \gg 1)$ with the operations $-$ and $\gg 1$. The 2^{nd} bit of y influences the 1^{st} output bit

of $\gg 1$, which in turn influences the 1st and 3rd output bits of $-$, among others. Hence, the shape abstraction for $x - (y \gg 1)$ is $[\blacktriangle, \blacksquare]$. This suggests that the composition of shape abstractions is related to the join $\sqcup$ of the shape lattice since $[\blacktriangle, \blacktriangle]$ is the shape abstraction of $-$, $[\blacktriangleright]$ is the shape abstraction of $\gg 1$, and $\blacktriangle \sqcup \blacktriangleright = \blacksquare$ is the smallest element greater than or equal to both $\blacktriangle$ and $\blacktriangleright$.

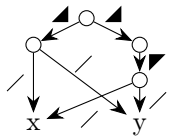


Fig. 2. Shapes of the running example.

To show this, assume that o_2 is connected to an arg-edge a_1 of shape s_1 of o_1. Now examine an arg-edge a_2 of o_2 that is in shape s_2. If $s_1 = /$, then the i-th input bit of a_1 affects at most the i-th output bit of o_1. Thus, if we replace these bits by those from the corresponding entry of $OA(o_2)$, a_2 is at most in the shape s_2 in $OA(o_1 \circ o_2)$. Similarly, if $s_1 = \blacktriangle$ and $s_2 = /$, then a_2 is at most in the shape $\blacktriangle$ in $OA(o_1 \circ o_2)$ because the i-th input bit of o_2 influences only the i-th output bit and o_1 spreads the i-th input bit only to the output bits $j \geq i$. An exhaustive analysis of all cases reveals that the shape of a_2 in $o_1 \circ o_2$ is at most $s_1 \sqcup s_2$. Thus, for a path $a_1, \ldots, a_n$ in P from the source to an input variable over arg-edges a_i of shape s_i, the total Path Shape $\mathrm{PS}_P(a_1, \ldots, a_n) = \bigsqcup_{1 \leq i \leq n} s_i$. The shape of an input variable v of P is the join of the path shapes of all paths that reach v. For example, the two paths that reach x in Fig. 2 have shapes $\blacktriangle \sqcup / = \blacktriangle$ and $\blacktriangle \sqcup \blacktriangleright \sqcup / = \blacksquare$, hence the shape of variable x is $\blacktriangle \sqcup \blacksquare = \blacksquare$. If there is only one unique path, we use a simplified notation $\mathrm{PS}_P(a_n) = \mathrm{PS}_P(a_1, \ldots, a_n)$.

If P has an input variable v with a shape in P (computed via OA) that is not greater or equal to the shape in f (computed via UA), there must be an output bit that is essential in some input bit in f but not in P. Hence, $UA(f) \not\subseteq OA(P)$. If there is no P such that all input variables have greater or equal shape in P than in f, there is no upper bound and thus the synthesis problem is infeasible.

4.2 Tree Upper Bounds

The program $(x \vee y) - ((x \oplus y) \gg 1)$ is not the only upper bound for our example synthesis task. Fig. 3 shows another upper bound with some special properties. First, some operations do not refer to an operand (denoted by "?"). Arbitrary values may be inserted here. As our infeasibility checker only uses the leaf shapes of a program and does not run it, we may keep these "loose ends" in our programs. Second, the result of every operation and every input variable is only used once. Hence, this program is formed like a tree. Does existence of an upper bound imply existence of such a tree upper bound? Luckily, we show below that whenever there is an upper bound program for a specification function f, there is also an upper bound whose underlying undirected graph (that results from replacing all directed edges $a \to b$ by a single undirected edge) is a tree. Hence, parallel arg-edges collapse to a single edge in the underlying undirected graph. Instead of a (hopefully failing) exhaustive search for an upper bound among arbitrary programs, it hence suffices to check the much smaller search space of tree programs that use each input variable only once. Such tree programs are not "real" programs in the sense

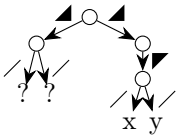


Fig. 3. Tree program.

that they can be executed since some operands are unconnected (as in Fig. 3). Their only purpose is to provide a simple skeleton that abstracts several real programs. In the following proof, we transform an arbitrary upper bound into a tree program. If the underlying undirected graph of a program is not a tree, it must contain a cycle. We reduce these cycles until they can be eliminated from the program. These transformations ensure that the shape of the input variables of the resulting program are at least as large as in the original program.

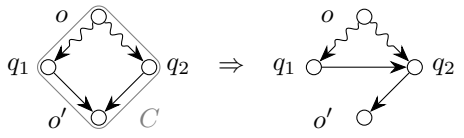
Lemma 2. *Suppose P is an upper bound of f. Then there is an upper bound P' that uses the same operations but whose underlying undirected graph is a tree.*

Proof. We construct a sequence of upper bounds $P_0, \dots, P_n$ such that $P_0 = P$ and $P_n =: P'$.

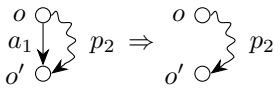
If for an arbitrary $i \geq 0$ the underlying undirected graph G_i of P_i is a tree, then $n := i$. Otherwise, G_i has a cycle C of minimal length with an operation o that has a minimal distance from the source among the operations in C. Then C contains exactly two non-parallel arg-edges a_1, a_2 of o with shapes s_1, s_2. Let p_1, p_2 be the (unique and distinct) paths in P_i from o to a common end o' that include a_1 resp. a_2 such that their union forms an orientation of C.

Cycle reduction: Assume that $|p_1| > 1$ and $|p_2| > 1$. Let q_1 (resp. q_2) be the operation that directly precedes o' on p_1 (resp. p_2). Since P_i is a DAG, P_i cannot contain both a path from q_1 to q_2 and from q_2 to q_1. Assume that P_i does not contain a path from q_2 to q_1 (the other case is similar). Then P_{i+1} results from the transformation shown in Fig. 4a. To show that P_{i+1} is also an upper bound, let p be an arbitrary path from the source to an arbitrary variable x in P_i. If p does not contain the edge (q_1, o'), p is also a path in P_{i+1}. Otherwise, we can obtain a corresponding path p' in P_{i+1} by replacing (q_1, o') by the two edges (q_1, q_2) and (q_2, o'). The edge (q_1, o') in P_i and (q_1, q_2) in P_{i+1} correspond to the same operand of q_1 and hence have the same shape. Let s be the shape of the operand corresponding to the edge (q_2, o'). Then $\mathrm{PS}_{P_{i+1}}(p') = \mathrm{PS}_{P_i}(p) \sqcup s \geq \mathrm{PS}_{P_i}(p)$. As this holds for all paths p and variables x, P_{i+1} is also an upper bound. G_{i+1} has a smaller cycle than the minimal C of G_i.

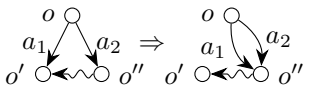
Cycle elimination, $|p_1| = 1$ ($|p_2| = 1$ is similar): Case 1, $s_1 \leq \mathrm{PS}_{P_i}(p_2)$: Define P_{i+1} by removing the arg-edge a_1 in P_i (Fig. 4b). Then P_{i+1} is also an upper



(a) Reduction of a minimal cycle C.



(b) Cycle elimination, case 1.



(c) Cycle elimination, case 3.

Fig. 4. Tree construction. ⇝➝: paths that may include arg-edges a_1 or a_2.

bound: For each path p over a_1 in P_i we get a path p' in both P_i and P_{i+1} by replacing a_1 with p_2 such that $\mathrm{PS}_{P_i}(p) \leq \mathrm{PS}_{P_i}(p')$. Hence, $\mathrm{PS}_{P_i}(p) \sqcup \mathrm{PS}_{P_i}(p') = \mathrm{PS}_{P_{i+1}}(p')$. P_{i+1} is an upper bound as the join of the shapes of all paths reaching a variable x is the same in P_i and P_{i+1}. G_{i+1} has fewer cycles than G_i. Case 2, $s_1 > \mathrm{PS}_{P_i}(p_2)$: Similar to case 1 but remove the last arg-edge of p_2 instead of a_1. Case 3, $s_1 \parallel \mathrm{PS}_{P_i}(p_2)$: Define P_{i+1} by reconnecting a_1 to the operation o'' that a_2 refers to (Fig. 4c). To show that P_{i+1} is an upper bound, let p be an arbitrary path from the source to an arbitrary variable x in P_i. If p does not contain a_1, p is also a path in P_{i+1}. Otherwise, there is a path p' to x in P_{i+1} that contains the reconnected a_1 and all edges except a_2 from p_2 since a_1 and p_2 have a common end in P_i. Similar to above, $\mathrm{PS}_{P_{i+1}}(p') \geq \mathrm{PS}_{P_i}(p)$ and hence the join of the shapes of all paths from the source to x is at least as large in P_{i+1} as in P_i. Since p and x are arbitrary, all variables are in at least the same shape in P_{i+1} as in P_i and P_{i+1} is also an upper bound. G_{i+1} has fewer cycles than G_i.

As these transformations keep the upper bound property and eventually remove all cycles, G_n is a tree. □

Note that our infeasibility checker never constructs a tree program from a general program. It only explores the search space of tree programs. It even suffices to explore tree programs without parallel edges as such edges can be transformed away without affecting the upper bound property: For parallel edges with shapes $s_1 \leq s_2$ it suffices to keep the edge for s_2. We omit the case of incomparable shapes as there are no such arg-edges in the synthesis problems of our evaluation in Sec. 5 and hence it is unlikely that they occur in practice.

4.3 Bipartite Matching of Variables to Usage Locations

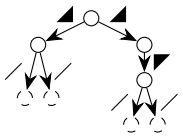


Fig. 5. Leaves program.

A (hopefully failing) exhaustive search for upper bounds among the tree programs examines many tree programs that differ only in the usages of the input variables. In the tree program in Fig. 3, we can, for example, swap x with y or one of the "?". In total, there are $\binom{4}{2} = 12$ ways to connect the given variables to this arrangement of operations. An exhaustive search would examine these 12 possibilities for each possible arrangement of the operations. To tune the search and avoid many redundant configurations, we simply omit the variables during the search. Hence, the search space shrinks to the space of *leaves programs* that use placeholder nodes (*leaves*) instead of variables (see Fig. 5). Then we view the problem of replacing the leaves by variables as a bipartite matching problem. Recall that in an upper bound each input variable must have a shape that is greater or equal to its respective shape in the specification function. To tailor our search towards upper bounds, a variable can only be matched to a leaf if this puts the variable in a shape required by an upper bound. In Fig. 5, for example, x may only be matched to one of the two rightmost leaves.

Consequently, we search for an upper bound using the operations alone and then obtain a bipartite matching from the variables to leaves that gives them at

least the *variable shapes* of the input variables of the specification function f. The multiset V holds all variable shapes of f. The path shape of the unique path from the root to a placeholder is its *leaf shape*. The multiset $\mathrm{L}(P)$ contains all leaf shapes of a leaves programs P. A leaves program is an upper bound if there is a matching of each variable v to a leaf ℓ whose shape is at least as large as the shape of v. If no leaves program admits such a matching, the synthesis problem is infeasible. It thus suffices to compute the size ν of a maximum matching, which can be done efficiently with the formula we give in Lemma 3.

As we did in the proof of Lemma 2, we shall later transform upper bounds, making use of the condition in Lemma 4 that guarantees that a second leaves program T can match at least as many variables as P. Hence, if P is an upper bound, so is T (needed in the search space reduction in Sec. 4.4). Finding a program with a maximum number of matched variables is an optimization problem yielding an optimum program. If this optimum program does not admit a matching of all variables, the synthesis problem is infeasible.

In the course of the search, we expand leaves programs by adding an operation. If an incomplete leaves program cannot be expanded to an optimum program, it may be ignored during the search. Lemma 5 extends Lemma 4 to also consider further expansions.

Some more notations: Restrictions of variable/leaf shapes to some set of shapes X: $\mathrm{V}(X) = \{x \in \mathrm{V} \mid x \in X\}$ and $\mathrm{L}(P, X) = \{x \in \mathrm{L}(P) \mid x \in X\}$; universe of shapes U; *downward closure* $\downarrow X = \{y \in U \mid \exists x \in X.\, x \geq y\}$ of a set of shapes X. Analogously, $\uparrow X = \{y \in U \mid \exists x \in X.\, x \leq y\}$. Leaf shape difference w.r.t. a set of shapes X: $\delta(P, T, X) = \{y \in \mathrm{L}(P) \setminus \mathrm{L}(T) \mid y \in X\}$. Missing proofs for the lemmas can be found in the artifact for the paper [13].

Lemma 3. *The maximum number of matched variables of a leaves program P is $\nu(P) = |\mathrm{L}(P)| - \max_X(|\mathrm{L}(P, \downarrow X)| - |\mathrm{V}(\downarrow X)|)$.*

To compute ν, we only need to consider those sets X with distinct $\downarrow X$, which are $\{◿\}, \{◢\}, \{◤\}, \{◢, ◤\}, \{■\}$. A single iteration that counts the leaf shapes for each of these sets then yields all $|\mathrm{L}(P, \downarrow X)|$. This is a lot easier than trying each possibility of assigning the variables to the leaves. Besides, the check whether a leaves program P is an upper bound reduces to $\nu(P) = |V|$.

Next, we relate the size of maximum matchings for two leaves programs. Intuitively, if a leaves program T has larger leaf shapes than P, it should match at least as many variables as P. The next lemma formalizes this insight.

Lemma 4. *Suppose P and T are leaves programs with $|\mathrm{L}(P)| = |\mathrm{L}(T)|$. If for all sets of shapes X, $|\delta(P, T, \uparrow X)| \leq |\delta(T, P, \uparrow X)|$, then $\nu(T) \geq \nu(P)$.*

If as many leaves of P as possible match to leaves of T with the same shape and if the remaining leaves of P match to larger leaves of T, Hall's Marriage Theorem provides the necessary condition for Lemma 4. This is handy if T is the result of some transformation on P because we can prove that the transformation preserves optimality if we can obtain such a matching by examining its steps.

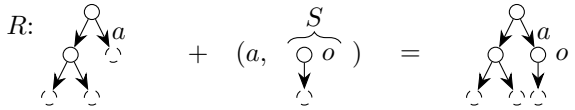


Fig. 6. Expansion of a leaves program (placeholder: ◡).

A leaves program R is a *root* of a leaves program P if R and P have the same root operation and R is an induced subtree of P. (R and P have the same structure except that the successor of an operation may be another operation in P but a leaf in R.) An exhaustive search can then expand R to get closer to P by replacing a leaf arg-edge a with a subtree S formed of several operations, see Fig. 6, notation: $R + (a, S)$. Suppose we can expand a leaves program R by either γ_1 or γ_2. If we choose the better expansion according to Lemma 4, is it possible that this choice must lead to a worse program after subsequent expansions? Lemma 5 shows that this is impossible:

Lemma 5. *If a leaves program P is rooted in $R + \gamma_1$ and there is another expansion γ_2 of R such that γ_2 comprises the same operations as γ_1 and for all sets of shapes X, $|\delta(R+\gamma_1, R+\gamma_2, \uparrow X)| \leq |\delta(R+\gamma_2, R+\gamma_1, \uparrow X)|$, then there is a P' rooted in $R+\gamma_2$ with $\nu(P') \geq \nu(P)$.*

4.4 Infeasibility Checks with Upper Bounds

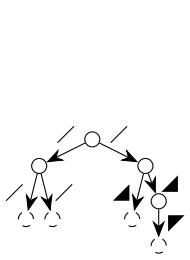
Although a (hopefully failing) exhaustive search constructs leaves programs only to get their leaf shapes, the order in which the search considers operations matters. For example, Fig. 7 shows an optimal leaves program if we pick the operations with ⟋ operand shapes first and the operation with ◤ operand shape last. This leaves program is, however, not an optimum program and hence no upper bound because it admits a matching of only one variable in contrast to Fig. 5. The reason is the position of the operations that provide ◢ and ◤ shapes. As these two operations together supply a $◢ \sqcup ◤ = ■$ shape, it intuitively makes sense to pick these operations first and the other operations that cannot further enlarge this shape later. We show below that there are always at most two operations that should be picked first (although we do not always know which). The remaining operations can then be considered in an arbitrary order.

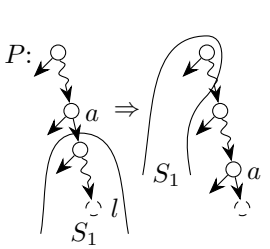
But let us first introduce three transformations T1 to T3 for leaves programs and conditions under which they do not reduce ν. We shall use T1 to T3 in the proofs below. Readers may wish to skip these on a cursory read.

As Fig. 8 shows, $\mathrm{T1}(P, a)$ removes a subtree S_1 connected to a and connects the remaining tree to an arbitrary leaf ℓ of S_1.

Lemma 6. *Suppose $a_1, \ldots, a_n$ is a path in a leaves program P. If for $1 \leq i < j \leq n$, $\mathrm{PS}_P(a_{j-1}) = ⟋ < \mathrm{PS}_P(a_j)$, then $\nu(\mathrm{T1}(P, a_i)) \geq \nu(P)$.*

$\mathrm{T2}(P, a_1, a_j)$ swaps the subtree S_1 connected to the arg-edge a_j with the subtree S_2 connected to a_1, see Fig. 9.





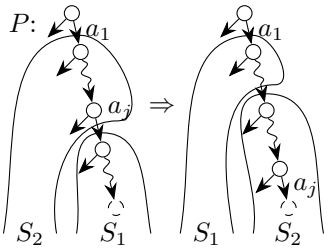


Fig. 7. No upper bound. **Fig. 8.** Effect of $\mathrm{T1}(P, a)$. **Fig. 9.** Effect of $\mathrm{T2}(P, a_1, a_j)$.

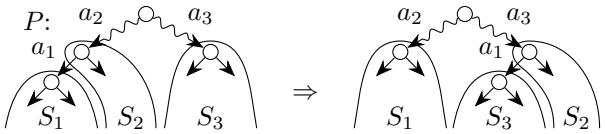


Fig. 10. Effect of $\mathrm{T3}(P, a_1, a_2, a_3)$.

Lemma 7. *Suppose $a_1, \ldots, a_n$ is a path in a leaves program P. If for $1 \leq j \leq n$, $\mathrm{PS}_P(a_1) = \mathrm{PS}_P(a_j)$, then $\nu(\mathrm{T2}(P, a_1, a_j)) \geq \nu(P)$.*

$\mathrm{T3}(P, a_1, a_2, a_3)$ removes the subtrees S_1, S_2, S_3 connected to arg-edges a_1, a_2, a_3 respectively. Then it connects S_2 to a_3, S_3 to a_1, and S_1 to a_2, see Fig. 10.

Lemma 8. *Suppose P is a program with arg-edges a_1, a_2, a_3 such that (a) $\mathrm{PS}_P(a_2) \leq \mathrm{PS}_P(a_3)$, (b) there is no path that contains both a_2 and a_3, (c) an operation o is connected to a_2, a_1 is an arg-edge of o, and $\mathrm{PS}_P(a_2) = \mathrm{PS}_P(a_1)$. Then $\nu(\mathrm{T3}(P, a_1, a_2, a_3)) \geq \nu(P)$.*

Lemma 5 already revealed that expanding only one leaf arg-edge for every leaf shape suffices because expanding a second arg-edge of the same shape with the same operations cannot yield a better program. Now we show that it even suffices to consider (almost) only a single arbitrary permutation of operations during the search for upper bounds: We first show that there is a leaves program consisting of at most two operations (a so-called *seed*) that can be expanded to an optimum solution. Second, we show that there is a seed that can be expanded to an optimum program with the remaining operations in an arbitrary order.

Lemma 9. *There is an optimum program O rooted in a seed D of at most two operations with $\max(\mathrm{L}(D)) = \max(\mathrm{L}(O))$.*

Proof. Let Q be an arbitrary optimum leaves program with root operation r. If $\max(\mathrm{L}(Q))$ is a subset of the set of arg-edge shapes of r, r alone satisfies the Lemma. Otherwise, there is a shape in $\max(\mathrm{L}(Q))$ that is not provided by r. If r has only arg-edges of shape $\diagup$, there is a path $a_1, \ldots, a_i$ in Q with $\mathrm{PS}_Q(a_{i-1}) = \diagup < \mathrm{PS}_Q(a_i)$. By Lemma 6, $\mathrm{T1}(Q, a_{i-1})$ is also optimum and is rooted in an operation with an arg-edge of non-$\diagup$ shape. Hence, assume that

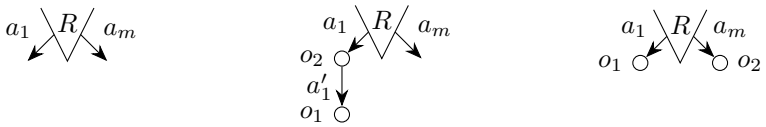


Fig. 11. Constellations in the proof of Lemma 10.

r has a non-$\diagup$ arg-edge a_r of shape s_r. This implies that there is exactly one shape s_m in $\max(\mathrm{L}(Q))$ that is not an arg-edge shape of r. There are three cases. (a) If Q has a path $a'_1, \ldots, a'_n$ with $\mathrm{PS}_Q(a'_1) = \mathrm{PS}_Q(a'_{n-1})$ and $\mathrm{PS}_Q(a'_n) = s_m$, then $\mathrm{T2}(Q, a'_1, a'_{n-1})$ is also optimum by Lemma 7 and r plus the operand of a'_1 form its seed. For the remaining cases, the missing shape s_m is $\blacksquare$ and Q has a path $a'_1, \ldots, a'_n$ and a $1 < i < n$ with $\mathrm{PS}_Q(a'_1) = \diagup$, $\mathrm{PS}_Q(a'_n) = \blacksquare$. (b) If $\mathrm{PS}_Q(a'_i) = s_r$, then the disjoint subtrees connected to a'_i and a'_r may be swapped without altering any leaf shape. For the resulting program, case (a) applies. (c) If $\mathrm{PS}_Q(a'_i) \parallel s_r$, then $\mathrm{T1}(P, a'_{i-1})$ is also optimum by Lemma 6. Moreover, the prefix of $\diagup$ arg-edges is removed from the path so that case (a) applies. □

If a seed D cannot be expanded to an optimum program by picking the operations in an arbitrary order, the next Lemma shows that there is a "stronger" seed (that we can systematically search for). In its proof, we try to swap the order of two operations in an optimum program. In almost all cases, we can connect the swapped operations so that the resulting leaves program is still optimum. Otherwise, the two swapped operations provide the "stronger" seed.

Lemma 10. *Suppose that O is an optimum program rooted in a seed D with $\max(\mathrm{L}(D)) = \max(\mathrm{L}(O))$ and that no optimum program O' has a leaf shape $s \in \max(\mathrm{L}(O'))$ that is larger than some shape in $\max(\mathrm{L}(O))$.*

Let R be a program, a_1 an arg-edge of R, and a_2 an arg-edge of $R + (a_1, o_1)$ such that R is rooted in D and O is rooted in $R + (a_1, o_1) + (a_2, o_2)$. If no arg-edges a'_2 of R and a'_1 of $R+(a'_2, o_2)$ exist such that $R+(a'_2, o_2)+(a'_1, o_1)$ is a root of an optimum program, then there is a seed D' that is the root of an optimum program with $\max(\mathrm{L}(D')) = \max(\mathrm{L}(O))$ and there is an $s_D \in \mathrm{L}(D)$ such that, for all leaf shapes $s_{D'} \in \mathrm{L}(D')$, $s_D < s_{D'}$.

Proof. Since $\max(\mathrm{L}(O)) = \max(\mathrm{L}(D))$, $\max(\mathrm{L}(R)) = \max(\mathrm{L}(O))$. Let $s_1 = \mathrm{PS}_O(a_1)$ and $s_2 = \mathrm{PS}_O(a_2)$. If there are arg-edges a'_2 of R and a'_1 of $R + (a'_2, o_2)$ such that $\mathrm{PS}_R(a'_2) = s_2$ and $\mathrm{PS}_{R+(a'_2,o_2)}(a'_1) = s_1$, then $\mathrm{L}(R + (a'_2, o_2) + (a'_1, o_1)) = \mathrm{L}(R + (a_1, o_1) + (a_2, o_2))$. Hence, $R + (a'_2, o_2) + (a'_1, o_1)$ can also be expanded to an optimum program by Lemma 5. Otherwise, there are two cases:

(1) $s_2 \notin \mathrm{L}(R)$. Since $\max(\mathrm{L}(R)) = \max(\mathrm{L}(O))$, R must have some arg-edge a_m of shape $s_m > s_2 > s_1$ (see Fig. 11, left). There are three sub-cases:

(1a) All arg-edges of o_2 have a larger shape than s_2. Then o_2 may also be connected to arg-edges of smaller shape than s_2. If o_2 is connected to a_1 of R and o_1 is connected to an arg-edge a'_1 of o_2 (see Fig. 11, center), the leaves provided by o_2 in $R + (a_1, o_1) + (a_2, o_2)$ are still present in the resulting program (since a_2 is an arg-edge of o_1, the shape of a'_1 is still present). For every leaf shape

contributed by o_1 in $R + (a_1, o_1) + (a_2, o_2)$, o_1 contributes a leaf that is at least as large in $R + (a_1, o_2) + (a'_1, o_1)$ because the shape of a'_1 is larger than s_1. By Lemma 5, this is also a root of an optimum program.

(1b) There is some arg-edge of o_2 with shape $t \leq s_2$. If o_2 is connected to a_m and o_1 is connected to a_1 (see Fig. 11, right; since $s_m > s_2 > s_1$, these are distinct arg-edges), the leaves provided by o_1 in $R + (a_1, o_1) + (a_2, o_2)$ (plus $s_2 \geq t$) are still present in the resulting program. For every leaf shape contributed by o_2 in $R + (a_1, o_1) + (a_2, o_2)$, o_2 contributes a leaf that is at least as large in $R+(a_m, o_2)+(a_1, o_1)$. Viewed as a matching, the leaf of a_m in $R+(a_1, o_1)+(a_2, o_2)$ and the leaf of a_2 in $R + (a_m, o_2) + (a_1, o_1)$ are unmatched. But for the arg-edge a_t of shape t, we have $\mathrm{PS}_{R+(a_1,o_1)+(a_2,o_2)}(a_t) = s_2$ and $\mathrm{PS}_{R+(a_m,o_2)+(a_1,o_1)}(a_t) = s_m$. Thus, the premise of Lemma 5 holds and $R + (a_m, o_2) + (a_1, o_1)$ is also the root of an optimum program.

(1c) The smallest arg-edge shape of o_2 is incomparable to s_2. Then $s_m = \blacksquare$ and $\max(\mathrm{L}(O)) = \{\blacksquare\}$. Also, since $s_m > s_2 > s_1$, $s_1 = \diagup$. If o_1 has an arg-edge $a_\diagup$ of shape $\diagup$, $\mathrm{T3}(O, a_\diagup, a_1, a_m)$ is also optimum by Lemma 8. Since o_1 and o_2 occur in shape s_m in $\mathrm{T3}(O, a_\diagup, a_1, a_m)$, there exists an expansion by o_2 followed by o_1 (as seen before).

Now, assume that no expansion by o_2 and then o_1 is a root of an optimum program. Hence, o_1 cannot have an arg-edge of shape $\diagup$. Since the program $o_1 + (a_2, o_2)$ can provide the shape $\blacksquare = s_m$, the program has no leaf of shape $\diagup$, and $s_1 = \diagup$, we know by Lemma 6 that this program is the root of the optimum program $\mathrm{T1}(O, a_1)$. Hence, $o_1 + (a_2, o_2)$ is the required seed as it has no leaf of shape $\diagup = s_1$ and consists of two operations.

(2) There is an arg-edge a'_2 of R with $\mathrm{PS}_R(a'_2) = s_2$ but $s_1 \notin \mathrm{L}(R + (a'_2, o_2))$. As $s_1 \in \mathrm{L}(R)$ by assumption, o_2 must be connected to the only arg-edge a'_2 with $\mathrm{PS}_R(a'_2) = s_2 = s_1$. If we connect o_1 to any arg-edge of o_2, each leaf of o_1 in this program is larger than or equal to its counterpart in $R + (a_1, o_1) + (a_2, o_2)$. Thus, by Lemma 5, this can also be expanded to an optimum program. □

We can simply try to expand all possible seeds and pick the best program because Lemmas 9 and 10 guarantee that this will be an optimum program. In our running example, we then eventually pick a seed consisting of the operations with shape abstractions [◢, ◢] and [◤] and grow the optimum program shown in Fig. 5 from this seed. This is the idea behind Algorithm 1.

Now we show that Algorithm 1 only flags synthesis problems as infeasible that are truly infeasible. Since we deal with leaves programs, we need to show that the algorithm only returns "infeasible" if there is no leaves program with a matching of all variables to some leaf. To compute the size of a maximum matching, Algorithm 1 uses the matching function ν that is defined in Lemma 3.

Theorem 1. *Algorithm 1 returns "infeasible" if and only if there is no leaves program P with $\nu(P) = |\,\mathrm{V}\,|$.*

Proof. By Lemmas 9 and 10, there is a seed D of at most two operations that can be expanded to an optimum program by all permutations of operations. The main loop (lines 2–10) considers all these seeds and hence considers D.

input : Operations Ops, Variable shapes V
output: Feasibility flag

```
1  B ← ∅;
2  foreach seed D of at most two operations do
3  |  S_0 ← {D}; ROps ← Ops \ {o | D contains operation o};
4  |  for i ← 1 to |ROps| do                                   // ops-loop
5  |  |  S_i ← ∅;
6  |  |  foreach T ∈ S_{i-1} do
7  |  |  |  foreach s ∈ L(T) do
8  |  |  |  |  a ←Leaf arg-edge with PS_T(a) = s;
9  |  |  |  |  S_i ← S_i ∪ (T + (a, ROps[i]));
10 |  B ← B ∪ {arg max_{T∈S_{|ROps|}} (ν(T))};
11 if ν(arg max_{T∈B}(ν(T))) = |V| then
12 |  return unknown
13 return infeasible
```

Algorithm 1. Checking the existence of an upper bound.

Suppose that a permutation π of operations can expand D and that in iteration i of the ops-loop (lines 4–9) some program T in $\mathcal{S}_{i-1}$ can be expanded to an optimum program. Then there is some arg-edge a of T such that $T + (a, o_{\pi_i})$ can be expanded to an optimum program. For all a' with $\mathrm{PS}_T(a') = \mathrm{PS}_T(a)$, $T + (a', o_{\pi_i})$ can also be expanded to an optimum program by Lemma 5. As the inner loop considers $\mathrm{PS}_T(a)$, it also expands an arg-edge of shape $\mathrm{PS}_T(a)$ and includes the expansion in $\mathcal{S}_i$. As every iteration of the ops-loop produces a program that can be expanded to an optimum program, $\mathcal{S}_{|ROps|}$ contains an optimum program O that line 10 adds to $\mathcal{B}$. Hence, Algorithm 1 returns "infeasible" if and only if $\nu(O) < |\,\mathrm{V}\,|$. □

4.5 Infeasibility Check with the 1 in the Zhegalkin Polynomial

As Sec. 1 shows, bit dependencies come in two flavors. For example, the highest output bit of an average of $\neg x$ and y behaves differently than the other bits. The reason is that only the Zhegalkin polynomial of the highest bit contains a non-zero coefficient $a_\emptyset$ (i.e., the Zhegalkin polynomial contains a 1). Hence, the presence of a 1 determines the flavor of an output bit.

Recall that our definition of an upper bound implies that its Zhegalkin polynomial for an output bit must contain a 1 if the same holds for the specification. If such a program does not exist, the synthesis problem is infeasible. If it exists, such a program may use two ways to realize a 1 in some output bit: First, certain operations (e.g., constants) provide some 1s. Second, some operations can *propagate* a 1 from an input bit to another output bit. Consider, for example, the program $2 + 2$. As the Zhegalkin polynomial of the 2^{nd} bit of 2 but only the 3^{rd} bit of the result is 1, the operation + can at least propagate the 2^{nd} to the 3^{rd} bit. An infeasibility checker may thus compute all possible propagations and then check if these suffice to let the sources of 1 reach the required output bits.

input : Bit width w, Operations Ops, Goal G
output: Feasibility flag

```
1  Srcs ← ∅; J ← {(i, i) | 1 ≤ i ≤ w};
2  foreach Operation o ∈ Ops do
3  |  NJ ← J;
4  |  foreach Propagation of o from bit j to i do
5  |  |  NJ ← NJ ∪ {(k, ℓ) | (k, j) ∈ J, (i, ℓ) ∈ J};
6  |  Srcs ← Srcs ∪ {i | 1 ≤ i ≤ w, i-th output bit of o contains 1};
7  |  J ← NJ;
8  if ∃i ∈ G. ∀j ∈ Srcs. (j, i) ∉ J then
9  |  return infeasible
10 return unknown
```

Algorithm 2. Checking the propagations of 1.

Although OA contains all possible propagations, it is too imprecise for this infeasibility checker. Consider an operation $o(x, y)$ with the Zhegalkin polynomial $x_1 \wedge y_2$ for the 1st output bit. Then $OA(o_1) = \{x_1, y_2\}$ but a 1 in y_2 can never reach the 1st output bit if x_1 does not also contain 1. Thus, o does not help to propagate 1 from the 2nd to the 1st bit. Luckily, we may use the following trick to overapproximate the propagations from other input bits: For each output bit i we compute $OA(o_i)$ with the i-th input bits of all its operands forced to 0. Then this $OA(o_i)$ contains the overapproximated input bits that can reach the i-th output bit independently of the i-th input bit of each operand.

Next, we combine the propagations of all operations. Let the goal set G be the set of i such that $1 \in UA(f_i)$ for a specification function f. To check if 1 can never reach all output bits in G, Algorithm 2 computes a set J of propagations. It then flags a synthesis problem as infeasible if it is impossible to propagate a 1 from its possible sources to the output bits in G.

Theorem 2. *If Algorithm 2 returns "infeasible", there is an $i \in G$ such that no program comprising the operations Ops has a* 1 *in the Zhegalkin polynomial of its i-th output bit.*

Proof. Suppose that Algorithm 2 returns "infeasible". Then there is an output bit $i \in G$ but J does not contain a propagation from a bit in $Srcs$ to i (lines 8–9).

We show by induction on $|Ops|$ that J holds at least all propagations of all programs using a subset of the operations Ops. Initially, J holds the propagations of the empty program. Suppose an operation $o \notin Ops$ propagates bit j to i. Let P be an arbitrary program using operations $Ops \cup \{o\}$. Then there are disjoint sets $A, B \subseteq Ops$ such that o depends on the operations in A and the operations in B depend on o in P. By the induction hypothesis, J holds all possible propagations for A and B. Thus, if A propagates bit k to j and B propagates bit i to ℓ, then line 5 adds (k, ℓ) to J. Hence, J holds at least all possible propagations.

Also, $Srcs$ contains the bit positions of all sources of 1. Thus, if J lacks a propagation from a bit in $Srcs$ to i, then no program comprising the operations Ops has a 1 in the Zhegalkin polynomial of its i-th output bit. □

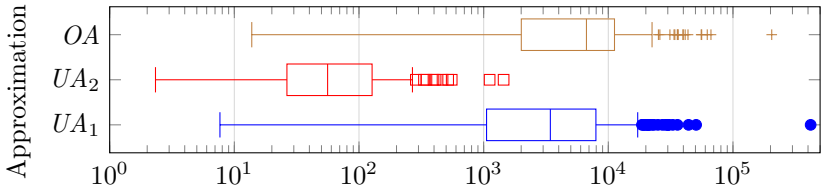


Fig. 12. Speedup of approximations over exact computation.

5 Evaluation

This section addresses three questions: (RQ1) How fast and accurate are our approximations compared with an exact computation? (RQ2) How much do algorithms 1 and 2 impact the solution time for synthesis problems? (RQ3) Do algorithms 1 and 2 detect hard infeasible problems?

As we are not aware of a benchmark of infeasible synthesis problems, we use specifications from four sources: The bit vector rewrite rules of Nötzli et al. [16], the code optimizations of Buchwald [3], the test cases in the public repository of Sasnauskas et al. [18] without undefined behavior, and the Syntax Guided Synthesis competition benchmark [1]. As we are restricted to bit vector specifications with a single equality constraint and without precondition, this yields 26+56+23+19 = 124 specifications. To turn them into synthesis problems that may be infeasible, we use them as input to an existing application in Secs. 5.2 and 5.3 that adds collections of available operations.

For comparative purposes, we implemented the synthesis method of Gulwani et al. [10] on top of the mature state-of-the-art SMT solver Yices 2.6.2 [9]. We implemented this synthesis method, our approximations, plus algorithms 1 and 2 straightforwardly in Java, without a lot of manual fine-tuning. All measurements ran on a computer equipped with an Intel i7-6920HQ processor and 32 GB of RAM running a Linux 5.7.11 kernel and Java 11.0.8+10.

5.1 Quality of the Approximations (RQ1)

The approximations UA_1, UA_2, and OA are one of the contributions of this paper. We examine how well they are suited for an analysis of the 124 specifications and how large the advantage of UA_2 is over UA_1.

To obtain the exact set of essential bits for each output bit, according to Sec. 2 we formulate for each synthesis problem multiple Yices queries "Does output bit i depend on input bit j?", one for each i and j.

Fig. 12 shows that computing UA_1 and OA is almost always at least three orders of magnitude faster than the exact computation. UA_2 is still 50 times faster in about half the cases. Fig. 13 shows that UA_1 and OA deliver perfect results for more than 50% of the specifications. UA_2 almost reaches 75% and can avoid the low agreement of UA_1 in some cases. UA_2 takes at most 120 ms for analyzing a single specification, whereas Yices requires up to 3 min, a disproportionate amount of time.

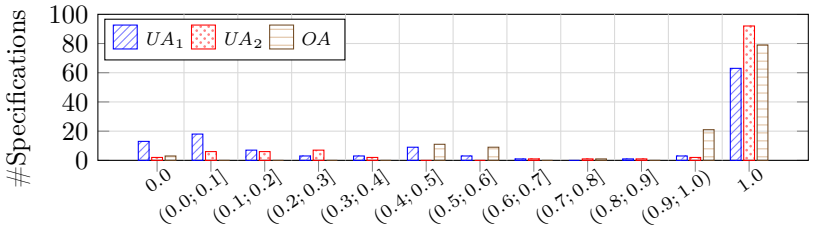


Fig. 13. Agreement with exact computation.

5.2 Impact of Algorithms 1 and 2 (RQ2)

Synthesizers for bit vector programs of bounded length usually detect infeasible synthesis problems. They benefit from our work if the runtime saving from avoiding the costly infeasibility proof exceeds the added runtime of our infeasibility check for all considered synthesis problems. This is what we evaluate here.

As a generator for synthesis problems and an example synthesizer we use Synapse [2], a technique that synthesizes optimal programs w.r.t. a user-specified cost model that assigns a cost to each operation. Such a problem lies at the heart of many applications [5, 17, 18, 19]. Later, we supply the generated synthesis problems to a second synthesizer. We feed Synapse the 124 specifications. Synapse uses a specification-specific set of operations (see below) and enumerates synthesis problems in order of increasing total cost of its collection of available operations. If a synthesis problem is infeasible or cannot be solved within the time limit, Synapse continues searching. If Synapse finds a program for a (feasible) synthesis problem, it stops and returns this program. Hence, all but the last synthesis problem are infeasible or time out. To reduce the risk that Synapse hits its time limit (which we set to 10 min), we use the enumeration scheme by Buchwald, Fried, and Hack [4]. We stop the enumeration for a single specification if Synapse cannot find a solution within 4 h. For simplicity, our cost model assigns a cost of 1 to all operations.

We set the specification-specific set of operations according to its benchmark source. For each source, this set comprises the union of operations used by programs implementing the specification functions from that source.[1] Since one of these sets of operations contains 9 custom operations, we also evaluate how well our checkers can handle operations beyond the well-known bit vector operations.

In total, Synapse generates 23,787 synthesis problems. For 2 specifications, Synapse hits the enumeration time limit of 4 h. Thus, 122 of the 23,787 synthesis problems are known to be feasible whereas 23,660 are infeasible and 5 are too difficult for Synapse to solve within the 10 min time limit. To avoid that our

[1] The benchmark sources provide a means to obtain the correct program. For example, some sources specify rewrite rules. Here, the left hand side corresponds to the specification function and the right hand side is an implementing program.

Table 1. Performance without and with algorithms 1 and 2 (runtimes are total times).

	Default		With pre-checking for infeasibility					
Synthesizer	Total	Timeouts	UA_2, OA	Alg. 1	Alg. 2	Total	Timeouts	Speedup
Synapse	13.1 h	5	11.1 min	3.9 s	207.3 s	8.8 h	5	32.93%
GJTV	490.1 h	1,818				455.6 h	1,730	7.04%

Table 2. Number of synthesis problems flagged as infeasible by algorithms 1 and 2 versus number of all synthesis problems for varying level of hardness.

Synthesizer	[0 ms; 100 ms]	(100 ms; 1 s]	(1 s; 10 s]	(10 s; 60 s]	(60 s; ∞)
Synapse	1,606/2,582	4,198/16,417	774/3,767	446/987	16/34
GJTV	2,884/5,900	1,663/6,328	1,646/4,227	497/2,875	350/4,457

results are specific to Synapse, we also feed the 23,787 synthesis problems to our implementation of the synthesis algorithm by Gulwani et al. [10] (GJTV).[2]

To evaluate the impact of our work, we apply algorithms 1 and 2 on *all* of these problems (including the feasible ones) and omit the costly synthesizer on the ones that our algorithms flag as infeasible, see Table 1. Algorithms 1 and 2 flag 5,261 resp. 3,013 (total: 7,040) synthesis problems as infeasible.

With our checkers, Synapse and GJTV take 33% resp. 7% less time. Although the improvement for GJTV is less strong (since the number of long-running problems is higher, see next section), our checkers can clearly speed up applications of bit vector synthesis. The total runtime of our checkers is 14.6 min, which is negligible compared with the runtimes of the synthesizers. Computing the approximations takes up to 170 times longer than running the algorithms. This shows the effectiveness of our search space reduction.

5.3 Hardness of Flagged Synthesis Problems (RQ3)

To the best of our knowledge, there is no objective measure of the hardness of a synthesis problem. But we may define the hardness of a problem for a specific synthesizer as its runtime on that problem. To do this, we use the runtimes of Synapse and GJTV for the 23,787 synthesis problems from Sec. 5.2. For easier presentation, we put the runtimes into five distinct bins, easy to hard, with boundaries at 100 ms, 1 s, 10 s, and 60 s. Then we investigate the proportion of synthesis problems that are flagged infeasible to all synthesis problems in a bin.

As Table 2 shows, the problems that our infeasibility checker flags as infeasible are spread over all bins. For Synapse, our checker flags almost half of the problems in the two hardest bins as infeasible. The percentage of flagged problems in these bins is lower for GJTV, but our checker can flag problems as infeasible on which GJTV hits the timeout. Thus, our work can also deal with hard synthesis problems.

[2] We observed that synthesizers for the class of Syntax Guided Synthesis problems perform poorly due to the necessary restrictions, as others noted before us [4].

5.4 Threats to Validity

Our results might be affected by implementation errors. We wrote more than $5,500$ test cases and performed billions of random tests to minimize this threat. Also, the selected specifications might not be representative. To mitigate this threat, we used four different sources for specifications. Last, the results might be specific to a synthesis algorithm or underlying solver. To make our results representative, we used two different synthesis algorithms that each uses a different solver (Z3 [15] and Yices). We make our implementation publicly available [13].

6 Related Work

Usually, infeasible problems are only addressed using domain-specific knowledge [4, 19] or by restricting the space of considered programs (e.g., to trees instead of DAGs [14]), so that infeasibility is detected faster at the cost of missing some feasible solutions.

To the best of our knowledge, Warren's right-to-left computability test [20, 21] is the only application of bit dependencies to the program feasibility problem. In our terminology, it states that, given a specification function, there is an implementing program composed of an arbitrary number of $+$, $-$, $\wedge$, $\vee$, and $\neg$ operations if and only if all variables are at most in shape $\blacktriangleleft$. Our work generalizes his test by supporting different kinds of shapes and arbitrary operations. Also, we present an algorithm to obtain the variable shapes.

Hu et al. [11] present an infeasibility check for unbounded synthesis problems that are constrained by a grammar. They reduce the infeasibility check to a program reachability problem that they tackle with a verification tool. For bounded bit vector synthesis problems as considered in this work, such a tool would be an SMT solver that the synthesis algorithms that we considered in Sec. 5 already rely on. Thus, the work of Hu et al. [11] and ours tackle disjoint problems.

Another infeasibility checker by Hu et al. [12] relies on a set of example input-output pairs to compute an abstraction of the set of terms that the grammar can generate via an extension of a dataflow analysis to grammars. Its performance depends on a good selection of examples. The checker only deals with unbounded synthesis problems in the theory of linear integer arithmetic. The choice of a good abstraction for bit vector synthesis tasks is an open problem.

7 Conclusion

The presented algorithms use bit dependencies to quickly flag some bit vector synthesis problems as infeasible. We also proposed approximations to compute the essential bits. As our techniques speed up synthesizers on infeasible problems by up to 33%, it is an interesting open problem to find further properties of bit vector functions to improve program synthesis.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A.: Syntax-Guided Synthesis. In: FMCAD'13: Formal Methods in Computer-Aided Design, pp. 1–8, Portland, OR (2013). Website: https://sygus.org
2. Bornholt, J., Torlak, E., Grossman, D., and Ceze, L.: Optimizing Synthesis with Metasketches. In: POPL'16: Principles of Programming Lang. Pp. 775–788, St. Petersburg, FL (2016). DOI: 10.1145/2837614.2837666
3. Buchwald, S.: Optgen: A Generator for Local Optimizations. In: CC'15: Compiler Construction, pp. 171–189, London, UK (2015). DOI: 10.1007/978-3-662-46663-6_9
4. Buchwald, S., Fried, A., and Hack, S.: Synthesizing an Instruction Selection Rule Library from Semantic Specifications. In: CGO'18: Code Generation and Optimization, pp. 300–313, Vienna, Austria (2018). DOI: 10.1145/3168821
5. Cowan, M., Moreau, T., Chen, T., Bornholt, J., and Ceze, L.: Automatic Generation of High-Performance Quantized Machine Learning Kernels. In: CGO'20: Code Generation and Optimization, pp. 305–316, San Diego, CA (2020). DOI: 10.1145/3368826.3377912
6. Crama, Y., and Hammer, P.L.: "Fundamental Concepts and Applications". In: Boolean Functions: Theory, Algorithms, and Applications. Cambridge, UK: Cambridge University Press, 2011, pp. 3–66. ISBN: 978-0-521-84751-3. DOI: 10.1017/CBO9780511852008.002.
7. Davey, B.A., and Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge, UK (2002)
8. Diestel, R.: Graph Theory. Springer, Berlin, Germany (2017). DOI: 10.1007/978-3-662-53622-3
9. Dutertre, B.: Yices 2.2. In: CAV'14: Computer Aided Verif. Pp. 737–744, Vienna, Austria (2014). DOI: 10.1007/978-3-319-08867-9_49
10. Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R.: Synthesis of Loop-free Programs. In: PLDI'11: Programming Lang. Design and Impl. Pp. 62–73, San Jose, CA (2011). DOI: 10.1145/1993498.1993506
11. Hu, Q., Breck, J., Cyphert, J., D'Antoni, L., and Reps, T.W.: Proving Unrealizability for Syntax-Guided Synthesis. In: CAV'19: Computer Aided Verif. Pp. 335–352, New York City, NY (2019). DOI: 10.1007/978-3-030-25540-4_18
12. Hu, Q., Cyphert, J., D'Antoni, L., and Reps, T.W.: Exact and Approximate Methods for Proving Unrealizability of Syntax-Guided Synthesis Problems. In: PLDI'20: Programming Lang. Design and Impl. Pp. 1128–1142, London, UK (2020). DOI: 10.1145/3385412.3385979
13. Kamp, M., and Philippsen, M.: *Artifact for "Approximate Bit Dependency Analysis to Identify Program Synthesis Problems as Infeasible"*, (2020). DOI: 10.5281/zenodo.4275482. https://doi.org/10.5281/zenodo.4275482. Oct. 2020
14. Mechtaev, S., Griggio, A., Cimatti, A., and Roychoudhury, A.: Symbolic Execution with Existential Second-Order Constraints. In: ESEC/FSE'18: European Softw. Eng. Conf. and Intl. Symp. Foundations of Softw. Eng. Pp. 389–399, Lake Buena Vista, FL (2018). DOI: 10.1145/3236024.3236049
15. Moura, L. de, and Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS'08: Tools and Algorithms for the Construction and Analysis of Sys. Pp. 337–340, Budapest, Hungary (2008). DOI: 10.1007/978-3-540-78800-3_24

16. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C., and Tinelli, C.: Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In: SAT'19: Theory and Applications of Satisfiability Testing, pp. 279–297, Lisbon, Portugal (2019). DOI: 10.1007/978-3-030-24258-9_20
17. Phothilimthana, P.M., Jelvis, T., Shah, R., Totla, N., Chasins, S., and Bodik, R.: Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. ACM SIGPLAN Not. **49**(6), 396–407 (2014). DOI: 10.1145/2666356.2594339
18. Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Lup, G., Taneja, J., and Regehr, J.: *Souper: A Synthesizing Superoptimizer*, (2018). arXiv: 1711.04422. Apr. 2018. Repository: https://github.com/google/souper
19. Van Geffen, J., Nelson, L., Dillig, I., Wang, X., and Torlak, E.: Synthesizing JIT Compilers for In-Kernel DSLs. In: CAV'20: Computer Aided Verif. Pp. 564–586, Los Angeles, CA (2020). DOI: 10.1007/978-3-030-53291-8_29
20. Warren Jr., H.S.: Functions Realizable with Word-Parallel Logical and Two's-Complement Addition Instructions. Commun. of the ACM **20**(6), 439–441 (1977). DOI: 10.1145/359605.359632
21. Warren Jr., H.S.: Hacker's Delight. Addison-Wesley, Upper Saddle River, NJ (2012)
22. Wegener, I.: The Complexity of Boolean Functions. B. G. Teubner, Stuttgart, Germany (1987)