Noname manuscript No.
(will be inserted by the editor)

Boolean Functional Synthesis: Hardness and Practical Algorithms

S. Akshay · Supratik Chakraborty · Shubham Goel · Sumith Kulal · Shetal Shah

the date of receipt and acceptance should be inserted later

Abstract Given a relational specification between Boolean inputs and outputs, Boolean functional synthesis seeks to synthesize each output as a function of the inputs such that the specification is met. Despite significant algorithmic advances in Boolean functional synthesis over the past few years, there are relatively small specifications that have remained beyond the reach of all state-of-the-art tools. In trying to understand this behaviour, we show that unless some hard conjectures in complexity theory are falsified, Boolean functional synthesis must generate large Skolem functions in the worst-case. Given this inherent hardness, what does one do to solve the problem? We present a two-phase algorithm, where the first phase is efficient in practice both in terms of time and size of synthesized functions, and solves a large fraction of our benchmarks. This phase is also guaranteed to solve the problem when the representation of the input specification satisfies some structural requirements. For those cases where the first phase doesn't suffice, we present a second phase of our synthesis algorithm that uses a special class of algorithms, called *expansion-based algorithms*, to generate correct Skolem functions. This may require exponential time and generate exponential-sized Skolem functions in the worst-case. Detailed experimental evaluation shows that our overall synthesis algorithm performs better than other techniques for a large number of benchmarks.

The authors wish to acknowledge funding support from DST/CEFIPRA/INRIA project EQuaVE and DST/SERB Matrices grant MTR/2018/000744 for S. Akshay, and from MHRD/IMPRINT-1/Project 5496(FMSAFE) for Supratik Chakraborty and Shetal Shah.

Most of this work was done when Shubham Goel and Sumith Kulal were at Indian Institute of Technology Bombay, India.

S. Akshay · Supratik Chakraborty · Shetal Shah
Indian Institute of Technology Bombay, India

Shubham Goel
University of California, Berkeley, USA

Sumith Kulal
Stanford University, USA

Keywords Boolean functional synthesis, Skolem functions, expansion-based algorithms

1 Introduction

Automatically synthesizing systems that always work as specified is one of the holy grails of computer-aided design. In many situations, it is unwieldy or even technically difficult to specify the desired behaviour of a system by expressing outputs as functions of inputs. Instead, it may be easier to specify the behaviour as a relation between inputs and outputs. Such specifications are also called *relational specifications*.

As an interesting example, consider a system with a single $2n$-bit unsigned integer input $\mathbf{Y}$, and two n-bit unsigned integer outputs $\mathbf{Z}_1$ and $\mathbf{Z}_2$. Suppose the relational specification is given as $F_{\mathsf{fact}}(\mathbf{Y}, \mathbf{Z}_1, \mathbf{Z}_2) \equiv ((\mathbf{Y} = \mathbf{Z}_1 \times_{[n]} \mathbf{Z}_2) \wedge (\mathbf{Z}_1 \neq 1) \wedge (\mathbf{Z}_2 \neq 1))$, where $\times_{[n]}$ denotes n-bit unsigned integer multiplication. This specification requires that $\mathbf{Z}_1$ and $\mathbf{Z}_2$ are non-trivial factors of $\mathbf{Y}$. Note, however, that if $\mathbf{Y}$ represents a prime number, there are no values of $\mathbf{Z}_1$ and $\mathbf{Z}_2$ that satisfy the specification. Therefore, we are interested in obtaining values of $\mathbf{Z}_1$ and $\mathbf{Z}_2$ that satisfy the specification, whenever possible. The easy part here is checking whether the specification is satisfiable for a given $\mathbf{Y}$, whereas the hard part is to synthesize concrete outputs as functions of given inputs. Significantly, the above specification can be encoded as a Boolean formula of size $\mathcal{O}(n^2)$ over the individual bits of $\mathbf{Y}$, $\mathbf{Z}_1$ and $\mathbf{Z}_2$. However, if we want to express $\mathbf{Z}_1$ and $\mathbf{Z}_2$ directly as Boolean functions of $\mathbf{Y}$, our task would be significantly harder. In fact, there are no known polynomial-sized Boolean functions that can express individual bits of $\mathbf{Z}_1$ and $\mathbf{Z}_2$ directly in terms of the individual bits of $\mathbf{Y}$[1]. This illustrates how relational specifications can be more natural and succinct than expressing outputs directly as functions of inputs. However, having conveniently represented specifications isn't good enough. We need to know *how difficult is it to synthesize systems whose behaviour is specified relationally?* In this paper, we investigate this question both from theoretical and practical perspectives.

Synthesizing Boolean functions from relational specifications has long been of interest to logicians and computer scientists. Formally, given a Boolean formula $F(\mathbf{Z}, \mathbf{Y})$ specifying a desired relation between inputs $\mathbf{Y}$ and outputs $\mathbf{Z}$, we wish to synthesize each output in $\mathbf{Z}$ as a function of the inputs $\mathbf{Y}$ such that $F(\mathbf{Z}, \mathbf{Y})$ is satisfied, whenever possible. Such functions have also been called *Skolem functions* in the literature [28, 23], and the quest for synthesizing Skolem functions and variants goes back long in history. In fact, Boole [8] and Löwenheim [32] studied variants of this problem in the context of finding most general unifiers. While these studies are theoretically elegant, implementations of the underlying techniques have been found to scale poorly beyond small problem instances [33]. More recently, synthesis of Boolean functions has found important applications in a wide range of contexts including reactive strategy synthesis [4, 47], certified QBF-SAT solving [39, 7, 35], automated program synthesis [44, 42], circuit repair and debugging [27], disjunctive decomposition of symbolic transition relations [46] and the like. This has

[1] Otherwise, we could efficiently factorize products of n-bit prime numbers, rendering cryptographic systems vulnerable to attacks.

spurred a lot of interest in developing practically efficient Boolean function synthesis algorithms. The resulting new generation of tools [23, 28, 1, 19, 45, 39, 38] have enabled synthesis of Boolean functions from much larger and more complex relational specifications than those that could be handled by earlier techniques, viz. [25, 7, 33].

In this paper, we study the Boolean functional synthesis problem from both theoretical and practical perspectives. Our investigation shows that unless some long-standing conjectures in computational complexity theory are falsified, Boolean functional synthesis must necessarily generate super-polynomial or even exponential-sized Skolem functions, thereby requiring super-polynomial or exponential time, in the worst-case. Therefore, it is unlikely that an efficient algorithm exists for solving all instances of Boolean functional synthesis. There are two ways to address this hardness in practice: (i) design algorithms that are provably efficient but may give "approximate" Skolem functions that are correct only on a fraction of all possible input assignments, or (ii) design an algorithm with worst-case exponential behaviour that provably solves all problem instances. In this work, we combine these approaches to design a two-phase synthesis algorithm. The first phase is provably efficient and suffices to solve a large fraction of our benchmarks. The second phase is invoked only if the first phase fails to synthesize Skolem functions for all outputs. The second phase of our algorithm adopts a counterexample-guided *expansion-based* approach, first proposed in [28] in the context of Boolean functional synthesis.

Our primary contributions can be summarized as follows.

1. We show that unless some long-standing complexity theoretic conjectures are falsified, Boolean functional synthesis must require super-polynomial time and space. Specifically, we show that unless $\mathsf{P} = \mathsf{NP}$, there exist problem instances where Boolean functional synthesis must take super-polynomial time. We also show that unless the Polynomial Hierarchy collapses to the second level, there exist problem instances that must generate super-polynomial sized Skolem functions. Finally, we prove that if the non-uniform exponential time hypothesis [15] holds, there exist problem instances that must generate exponential sized Skolem functions, thereby also requiring at least exponential time.
2. We present a new two-phase algorithm for Boolean functional synthesis.
   (a) Phase 1 of our algorithm generates candidate Skolem functions of size polynomial in the input specification. This phase makes polynomially many calls to an NP oracle (SAT solver in practice). Hence it directly benefits from the progess made by the SAT solving community, and is efficient in practice. Our experiments indicate that phase 1 suffices to solve a large majority of publicly available benchmarks.
   (b) However, there are indeed cases where the first phase is not enough. In such cases, the first phase provides good candidate Skolem functions as starting points for the second phase. In the second phase, our algorithm starts from these candidate Skolem functions, and uses an iterative approach to rectify erroneous Skolem functions. We define a class of algorithms called *expansion-based algorithms* for doing this, and present a hybrid algorithm that combines three different expansion-based algorithms. The sizes of the correct Skolem functions generated by this phase may be exponential in the

worst-case. This blow-up is unlikely to be avoidable, thanks to our hardness results.

3. We analyze the surprisingly good performance of the first phase (especially in light of the theoretical hardness results) and show a sufficient condition on the structure of the input representation that guarantees correctness of the first phase. Interestingly, popular representations like ROBDDs [12] give rise to input structures that satisfy this condition.
4. We conduct an extensive set of experiments over a variety of benchmarks, and show that our algorithm performs favourably vis-a-vis state-of-the-art algorithms for Boolean functional synthesis.

Related work The literature contains several early theoretical studies on variants of Boolean functional synthesis [8, 32, 18, 9, 34, 6]. More recently, researchers have tried to build practically efficient synthesis tools that scale to medium or large problem instances. In [23], Skolem functions for $\mathbf{Z}$ are extracted from a specific type of proof of validity of $\forall\mathbf{Y}\exists\mathbf{Z}\,F(\mathbf{Z},\mathbf{Y})$. While this works exceptionally well with short proofs of validity, it doesn't work when $\forall\mathbf{Y}\exists\mathbf{Z}\,F(\mathbf{Z},\mathbf{Y})$ is not valid. Specifications of the latter type are also called *unrealizable*. Despite the nomenclature, as our non-trivial factorization example shows, it is often important and useful to synthesize Skolem functions even for unrealizable specifications.

Inspired by the spectacular effectiveness of conflict-driven clause learning (CDCL) SAT solvers [41], an incremental determinization technique for Skolem function synthesis was proposed in [38], and subsequently developed further in [40, 37].

In [25, 46], a synthesis approach based on iterated compositions was proposed. Unfortunately, as has been noted in [28, 19], composition based synthesis approaches do not scale well to large benchmarks. A recent work [19] adapts the composition-based approach to work with ROBDDs, which can be represented compactly if we know the optimum variable ordering. For factored specifications, i.e, specifications that are conjunctions of sub-specifications, ideas from symbolic model checking using implicitly conjoined ROBDDs have been used to enhance the scalability of ROBDD-based synthesis further in [45].

In the genre of counterexample guided abstraction refinement (CEGAR) techniques, [28] showed how CEGAR can be used to synthesize Skolem functions from factored specifications. The key idea here is to start with initial easy-to-compute abstractions of Skolem functions and refine them iteratively using counterexamples generated by invoking a state-of-the-art SAT solver. Subsequently, a compositional and parallel technique for Skolem function synthesis from arbitrary specifications represented using and-inverter graphs (AIGs) was presented in [1]. The second phase of the synthesis algorithm proposed in this paper builds on some of this work.

An approach based on identifying and separating input and output components of a specification was proposed in [14]. While this approach doesn't perform as well as some other state-of-the-art approaches, it is able to solve some hard synthesis benchmarks, for which other state-of-the-art tools fail within reasonable resource constraints. Recently, a Boolean functional synthesis technique that leverages constrained sampling and machine learning to arrive at initial approximations of Skolem functions, and then iteratively repairs these approximations using counterexamples, was presented in [21]. This technique has been reported to outperform most existing Boolean functional synthesis techniques. However, since this work was published after the current paper was submitted and reviewed, we simply mention it here without using it for our experimental studies.

In addition to the above techniques, template-based [44] and sketch-based [43] approaches have been found to be effective for synthesis when we have information about the set of candidate solutions. In the absence of such information, however, these techniques are known not to perform well. On a related note, a framework for functional synthesis that reasons about some unbounded domains such as integer arithmetic, was proposed in [31].

2 Notations and Problem Statement

A Boolean formula $F(v_1, \ldots v_p)$ is a syntactic object constructed according to the rules of propositional logic, that represents a mapping from $\{0, 1\}^p$ to $\{0, 1\}$ under the standard semantics of propositional logic. For notational convenience, we use F to also refer to the semantic mapping represented by F when there is no confusion. The set of variables $\{v_1, \ldots v_p\}$ in F is called the *support* of F, and denoted $\mathsf{sup}(F)$. A *literal* is either a variable or its complement. We use $F|_{v_i=0}$ (resp. $F|_{v_i=1}$) to denote the positive (resp. negative) cofactor of F with respect to v_i, i.e. F with the variable v_i set to 0 (resp. 1). A *satisfying assignment* of F is a mapping of variables in $\mathsf{sup}(F)$ to $\{0, 1\}$ such that the semantic mapping represented by F evaluates to 1 under this assignment. If F has a satisfying assignment, we say that F is *satisfiable*; otherwise, F is said to be *unsatisfiable*. If every mapping of $\mathsf{sup}(F)$ to $\{0, 1\}$ is a satisfying assignment of F, we say that F is *valid*. If π is a satisfying assignment of F, we write $\pi \models F$ and use $\pi[v_i]$ to denote the value assigned to $v_i \in \mathsf{sup}(F)$ by π. Let $\mathbf{V} = (v_{i_1}, v_{i_2}, \ldots v_{i_j})$ be a sequence of variables in $\mathsf{sup}(F)$. We use $\pi{\downarrow_{\mathbf{V}}}$ to denote the projection of π on $\mathbf{V}$, i.e. the sequence $(\pi[v_{i_1}], \pi[v_{i_2}], \ldots \pi[v_{i_j}])$.

A Boolean function $\psi(u_1, \ldots u_q)$ is a mapping from $\{0, 1\}^q$ to $\{0, 1\}$, and may be represented in various ways. For purposes of this paper, we assume that every Boolean formula and Boolean function is represented as a rooted directed acyclic graph (DAG), with internal nodes labeled by Boolean operators and leaves labeled by input/output literals and Boolean constants. If the operator labeling an internal node N has arity k, we assume that N has k ordered children. Each node N in such a DAG represents a Boolean formula (resp. function) $\Phi(N)$ that is inductively defined as follows. If N is a leaf, $\Phi(N)$ is the literal labeling N. If N is an internal node labeled by op with arity k, and if the ordered children of N are $c_1, \ldots c_k$, then $\Phi(N)$ is $\mathsf{op}(\Phi(c_1), \ldots \Phi(c_k))$. A DAG with root R is said to represent the formula (resp. function) $\Phi(R)$. Note that popular DAG representations of Boolean formulas and functions, such as and-inverter graphs (AIGs [22, 30]), reduced ordered binary decision diagrams (ROBDDs [12]) and Boolean circuits, are either already in this representation or can be easily converted to this representation.

A Boolean formula is said to be represented in *negation normal form (NNF)* if (i) the only operators used in the representation are conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$), and (ii) negation is applied only to variables. Every Boolean formula can be converted to a semantically equivalent formula in NNF, in which the internal nodes are labeled with $\wedge$ and $\vee$, and leaves are labeled with literals. We use $|F|$ to denote the number of nodes in a DAG represention of F. In this paper, we use and-inverter graphs, or AIGs, as the initial representation of specifications. Given an AIG with t nodes, an equivalent NNF representation of size $\mathcal{O}(t)$ can be constructed in $\mathcal{O}(t)$ time. Henceforth, we will assume that every Boolean formula is in NNF, unless specified otherwise.

Let N be a node in a DAG representation of a Boolean formula F in NNF. We use $lits(N)$ to denote the set of literals labeling leaves that have a path from N in the DAG representation. We also use $atoms(N)$ to denote the underlying set of variables in $\mathsf{sup}(F)$ that appear in $lits(N)$. For each $\wedge$-labeled internal node N in the DAG of F with children $c_1, \ldots c_k$, if $atoms(c_r) \cap atoms(c_s) = \emptyset$ for all distinct $r, s \in \{1, \ldots k\}$, then F is said to be in *decomposable negation normal form* or DNNF [17]. While DNNF formulas enjoy many nice properties [17], a weaker form turns out to be useful for purposes of synthesis. Specifically, for each $\wedge$-labeled internal node N, suppose $c_1, \ldots c_k$ are its children, and $lits(c_r) \cap \{\neg \ell \mid \ell \in lits(c_s)\} = \emptyset$ for every distinct $r, s \in \{1, \ldots k\}$. Then F is said to be in *weak decomposable NNF*, or wDNNF. Note that every DNNF formula is also a wDNNF formula.

We say a *literal* l is *pure* in F iff the NNF representation of F has a leaf labeled l, but no leaf labeled $\neg l$. F is said to be *positive unate* in $v_i \in \mathsf{sup}(F)$ iff $F|_{v_i=0} \Rightarrow F|_{v_i=1}$. Similarly, F is said to be *negative unate* in v_i iff $F|_{v_i=1} \Rightarrow F|_{v_i=0}$. Finally, F is *unate* in v_i if it is either positive unate or negative unate in v_i. A formula that is not unate in $v_i \in \mathsf{sup}(F)$ is said to be *binate* in v_i.

Throughout this paper, we use $\mathbf{Z} = (z_1, \ldots z_n)$ to denote a sequence of Boolean outputs, and $\mathbf{Y} = (y_1, \ldots y_m)$ to denote a sequence of Boolean inputs. The *Boolean functional synthesis* problem, henceforth denoted BFnS, asks: *given a Boolean formula* $F(\mathbf{Z}, \mathbf{Y})$ *specifying a relation between inputs* $\mathbf{Y}$ *and outputs* $\mathbf{Z}$*, determine Boolean functions* $\mathbf{\Psi} = (\psi_1(\mathbf{Y}), \ldots \psi_n(\mathbf{Y}))$ *such that* $F(\mathbf{\Psi}, \mathbf{Y})$ *evaluates to true for every value of* $\mathbf{Y}$ *for which* $\exists \mathbf{Z} F(\mathbf{Z}, \mathbf{Y})$ *holds.* Thus, $\forall \mathbf{Y}\, (\exists \mathbf{Z}\, F(\mathbf{Z}, \mathbf{Y}) \Leftrightarrow F(\mathbf{\Psi}, \mathbf{Y}))$ must be rendered valid. The function ψ_i is called a *Skolem function* for z_i in F, and $\mathbf{\Psi} = (\psi_1, \ldots \psi_n)$ is called a *Skolem function vector* for $\mathbf{Z}$ in F. As with all Boolean functions in this paper, Skolem functions are assumed to be represented as DAGs with non-leaf nodes labeled by $\wedge$, $\vee$ and $\neg$.

For $1 \le i \le j \le n$, let $\mathbf{Z}_i^j$ denote the sub-sequence $(z_i, z_{i+1}, \ldots z_j)$ and let $F^{(i-1)}(\mathbf{Z}_i^n, \mathbf{Y})$ denote $\exists \mathbf{Z}_1^{i-1} F(\mathbf{Z}_1^{i-1}, \mathbf{Z}_i^n, \mathbf{Y})$. It has been argued in [25, 26, 28, 19, 1] that given a relational specification $F(\mathbf{Z}, \mathbf{Y})$, the BFnS problem can be solved by first imposing a linear order on the outputs, say $z_1 \prec z_2 \cdots \prec z_n$, and then synthesizing a function $\psi_i(\mathbf{Z}_{i+1}^n, \mathbf{Y})$ for each z_i such that $F^{(i-1)}(\psi_i, \mathbf{Z}_{i+1}^n, \mathbf{Y}) \Leftrightarrow \exists z_i F^{(i-1)}(z_i, \mathbf{Z}_{i+1}^n, \mathbf{Y})$. Once all such functions ψ_i are obtained, one can substitute ψ_{i+1} through ψ_n for z_{i+1} through z_n respectively, in ψ_i to obtain a Skolem function for z_i as a function of only $\mathbf{Y}$. We adopt this approach, and therefore focus on synthesizing ψ_i in terms of $\mathbf{Z}_{i+1}^n$ and $\mathbf{Y}$.

The following definitions, adapted from [28, 25], play a key role in this paper.

Definition 1 Given $F(\mathbf{Z}, \mathbf{Y})$ and an ordering $z_1 \prec z_2 \cdots \prec z_n$, let ${\Delta_i}^F(\mathbf{Z}_{i+1}^n, \mathbf{Y})$ denote $\neg \exists \mathbf{Z}_1^{i-1}\ F(\mathbf{Z}_1^{i-1}, 0, \mathbf{Z}_{i+1}^n, \mathbf{Y})$, and ${\Gamma_i}^F(\mathbf{Z}_{i+1}^n, \mathbf{Y})$ denote $\neg \exists \mathbf{Z}_1^{i-1} F(\mathbf{Z}_1^{i-1}, 1, \mathbf{Z}_{i+1}^n, \mathbf{Y})$. When F is clear from the context, we often omit mentioning it and write Δ_i and Γ_i instead of ${\Delta_i}^F$ and ${\Gamma_i}^F$ respectively.

Note that if Δ_i (resp. Γ_i) evaluates to 1 for a certain assignment to $\mathbf{Z}_{i+1}^n$ and $\mathbf{Y}$, then F cannot be satisfied if the Skolem function for z_i evaluates to 0 (resp. 1) for the same assignment. From [28, 25], we know that a function ψ_i is a Skolem function for z_i iff it satisfies ${\Delta_i}^F \Rightarrow \psi_i \Rightarrow \neg {\Gamma_i}^F$. It is also easy to see that both Δ_i and $\neg \Gamma_i$ serve as Skolem functions for z_i in F.

3 Complexity-theoretical limits

It is easy to see that BFnS can be solved in EXPTIME. Indeed a naive solution would be to enumerate all possible values of $\mathbf{Y}$ and invoke a SAT solver to find values of $\mathbf{Z}$ corresponding to each valuation of $\mathbf{Y}$ that makes $F(\mathbf{Z}, \mathbf{Y})$ true. This requires worst-case time exponential in the number of inputs and outputs, and may produce Skolem functions of size exponential in the number of inputs. We now ask if it is possible to do better.

Theorem 1 *1. Unless* P = NP*, there exist problem instances where any algorithm for* BFnS *must take super-polynomial time.*
2. Unless $\Sigma_2^{\mathsf{P}} = \Pi_2^{\mathsf{P}}$*, there exist problem instances where any algorithm for* BFnS *must generate super-polynomial sized Skolem functions*
3. Unless the non-uniform exponential-time hypothesis (or $\mathsf{ETH}_{\mathsf{nu}}$) *fails, there exist problem instances where any algorithm for* BFnS *must generate exponential sized Skolem functions.*

Before presenting the proof, a few points are worth noting. Violation of the assumption in the first statement implies a complete collapse of the Polynomial Hierarchy (PH), while violation of that in the second statement implies a collapse of PH to the second level. Whether either of these are possible remain long-standing open questions, although it is widely believed that the PH doesn't collapse. Furthermore, since a lower bound of the *size* of Skolem functions translates to a lower bound of the *time* taken to compute these functions, the second and third statements also imply conditional super-polynomial and exponential, respectively, lower bounds of time complexity.

The exponential-time hypothesis ETH [24] and its strengthened version – the non-uniform exponential-time hypothesis $\mathsf{ETH}_{\mathsf{nu}}$ [15]– are unproven computational hardness assumptions that have been used to show that several classical decision, functional and parametrized NP-complete problems are unlikely to have sub-exponential algorithms. As remarked in [15], the non-uniform variant is also widely believed to be true, with many results carrying over from the uniform setting. Formally, $\mathsf{ETH}_{\mathsf{nu}}$ states[2] that there is no family of algorithms (one for each input-size n) that can solve the n-variable instance of 3-SAT in sub-exponential time (i.e., in time $2^{o(n)}$).

Proof Part *1.* follows from the easy observation that propositional satisfiability can be reduced to BFnS where there are no inputs. Formally, consider an instance of 3-SAT where we ask if $\exists \mathbf{Z}\, F(\mathbf{Z})$ is true. This can be seen as an instance of BFnS where $\mathbf{Y}$ is empty. That is, given $F(\mathbf{Z})$, we wish to synthesize the Skolem function vector $\mathbf{\Psi}$, such that $\exists \mathbf{Z} F(\mathbf{Z}) \Leftrightarrow F(\mathbf{\Psi})$. In other words, $F(\mathbf{\Psi}) = 1$ iff $F(\mathbf{Z})$ is satisfiable. Now if $\mathbf{\Psi}$ can be synthesized in polynomial time, then it can at most be poly-sized and hence $F(\mathbf{\Psi})$ can be evaluated in polynomial time. Thus, as a consequence we obtain P = NP.

Consider an n-variable instance of the 3-CNF SAT problem $\varphi(\mathbf{Z})$, where $|\mathbf{Z}| = n$. As 3-SAT $\in$ NP, by definition of class NP, it has a polynomial time verifier. This implies that there is a polynomial size circuit C, which takes as inputs an encoding

[2] We use the standard definition for $\mathsf{ETH}_{\mathsf{nu}}$ see e.g., [15, 20]. We note however that in [16] the authors consider an alternate definition of this notion.

of the formula φ, say enc(φ) and witness assignment $\pi \in \{0,1\}^n$ and evaluates to 1 iff π is a satisfying assignment for φ. Since φ is a 3-CNF formula, enc(φ) has size $O(p(n))$ where $p(.)$ is a polynomial. This implies that for every $n > 0$, there is a polynomial size verifier circuit C_n and a corresponding Boolean formula $F_n(\mathbf{Z}, \mathbf{Y})$ with $|\mathbf{Z}| = n, |\mathbf{Y}| = p(n)$. Thus, we obtain an instance of BFnS, $F_n(\mathbf{Z}, \mathbf{Y})$.

- Now, for Part *2.*, if the Skolem functions synthesized $\mathbf{\Psi}(\mathbf{Y})$ are of size polynomial in n, $F_n(\mathbf{\Psi}(\mathbf{Y}), \mathbf{Y})$ would also be of size polynomial in n. Therefore for every 3-CNF formula $\varphi(\mathbf{Z})$ on n variables, satisfiability of φ can be decided by setting $\mathbf{Y} = \text{enc}(\varphi)$ in $F_n(\mathbf{\Psi}(\mathbf{Y}), \mathbf{Y})$. Thus, we obtain a solution for n-variable instance of 3-SAT using polynomial-sized circuits. Recall that problems that can be solved using polynomial-sized circuits are said to be in the class PSIZE (equivalently called P/poly). Now since 3-SAT is NP-complete, it follows that NP $\subseteq$ P/poly. By the Karp-Lipton Theorem [29], this implies that $\Sigma_2^{\mathsf{P}} = \Pi_2^{\mathsf{P}}$, which implies that the PH collapses to the second level.
- Similarly, for Part *3.*, if $\mathbf{\Psi}(\mathbf{Y})$ is of size $2^{o(n)}$, then $F(\mathbf{\Psi}(\mathbf{Y}), \mathbf{Y})$ will also be of size $2^{o(n)}$. In other words, we can evaluate this function in sub-exponential time $2^{o(n)}$ and thus solve the n-variable 3-SAT instance in time $2^{o(n)}$, thus violating $\mathsf{ETH}_{\mathsf{nu}}$. Note that since the circuits for the Skolem functions can vary with input lengths, we may have different algorithms for different input sizes. Hence we have to appeal to the non-uniform variant of ETH. $\square$

Theorem 1 implies that efficient algorithms for BFnS are unlikely. We therefore propose a two-phase algorithm to solve BFnS in practice. The first phase runs in polynomial time relative to an NP-oracle and generates polynomial-sized "approximate" Skolem functions. We show that under certain structural restrictions on the NNF representation of F, the first phase always returns correct Skolem functions. However, these structural restrictions may not always be met. An NP-oracle can be used to check if the functions computed by the first phase are indeed correct Skolem functions. In case they aren't, we proceed to the second phase of our algorithm that may take exponential time in the worst-case, but has been empirically found to work well in practice.

4 Opportunistic polynomial-sized synthesis

The first phase of our algorithm assumes access to an NP oracle (a SAT-solver in practice) and makes polynomially many calls to it. Given the spectacular improvements in SAT solving performance over the past few decades, our goal in this phase is to design an algorithm that achieves efficiency in practice while synthesizing Skolem functions that are polynomial-sized, whenever possible. To do so, we start by first processing the unate output variables in the input specification.

Proposition 1 *If $F(\mathbf{Z}, \mathbf{Y})$ is positive (resp. negative) unate in z_i, then $\psi_i = 1$ (resp. $\psi_i = 0$) is a correct Skolem function for z_i.*

Proof Recall that F is positive unate in z_i means $F|_{z_i=0} \Rightarrow F|_{z_i=1}$. It follows that $\exists z_i F \Leftrightarrow (F|_{z_i=0} \vee F|_{z_i=1}) \Leftrightarrow F|_{z_i=1}$. Hence, 1 is indeed a correct Skolem function for z_i in F. The proof for negative unateness follows along similar lines. $\square$

The above result gives us a way to identify outputs z_i for which a Skolem function can be easily computed. Note that if z_i (resp. $\neg z_i$) is a pure literal in F, then F is positive (resp. negative) unate in z_i. However, the converse is not necessarily true. In general, a semantic check is necessary to test for unateness. In fact, it follows from the definition of unateness that F is positive (resp. negative) unate in z_i iff the formula η_i^+ (resp. η_i^-) defined below is unsatisfiable.

$$\eta_i^+ = F(\mathbf{Z}_1^{i-1}, 0, \mathbf{Z}_{i+1}^n, \mathbf{Y}) \wedge \neg F(\mathbf{Z}_1^{i-1}, 1, \mathbf{Z}_{i+1}^n, \mathbf{Y}). \tag{1}$$

$$\eta_i^- = F(\mathbf{Z}_1^{i-1}, 1, \mathbf{Z}_{i+1}^n, \mathbf{Y}) \wedge \neg F(\mathbf{Z}_1^{i-1}, 0, \mathbf{Z}_{i+1}^n, \mathbf{Y}). \tag{2}$$

Note that each such check involves a single invocation of an NP-oracle, and a variant of this unateness check has been used in [5].

If F is binate in an output z_i, Proposition 1 doesn't help in synthesizing ψ_i. Towards synthesizing Skolem functions for such outputs, recall the definitions of Δ_i and Γ_i from Section 2. Clearly, if we can compute these functions, we can solve BFnS. While computing Δ_i and Γ_i *exactly* for all z_i is unlikely to be efficient in general (in light of Theorem 1), we show that polynomial-sized "good" approximations of Δ_i and Γ_i can indeed be computed efficiently. As our experiments show, these approximations are good enough to solve BFnS for several benchmarks.

Definition 2 Given a relational specification $F(\mathbf{Z}, \mathbf{Y})$, we use $\widehat{F}(\mathbf{Z}, \overline{\mathbf{Z}}, \mathbf{Y})$ to denote the Boolean formula obtained by first representing F in NNF, and then replacing every occurrence of $\neg z_i$ ($z_i \in \mathbf{Z}$) in the NNF representation with a fresh variable $\overline{z_i}$. The formula $\widehat{F}(\mathbf{Z}, \overline{\mathbf{Z}}, \mathbf{Y})$ is called the *positive form* of the specification $F(\mathbf{Z}, \mathbf{Y})$.

Example 1 Consider the specification $F(\mathbf{Z}, \mathbf{Y}) = (z_1 \vee y_1) \wedge (\neg z_1 \vee \neg z_2) \wedge (z_2 \vee \neg y_2) \wedge (\neg z_2 \vee \neg z_3 \vee \neg y_1) \wedge (z_3 \vee y_1) \wedge (\neg z_3 \vee y_2)$. The positive form is $\widehat{F}(\mathbf{Z}, \overline{\mathbf{Z}}, \mathbf{Y}) = (z_1 \vee y_1) \wedge (\overline{z_1} \vee \overline{z_2}) \wedge (z_2 \vee \neg y_2) \wedge (\overline{z_2} \vee \overline{z_3} \vee \neg y_1) \wedge (z_3 \vee y_1) \wedge (\overline{z_3} \vee y_2)$. $\square$

The following are easy consequences of Definition 2.

Proposition 2 *(a)* $\widehat{F}(\mathbf{Z}, \overline{\mathbf{Z}}, \mathbf{Y})$ *is positive unate in both* $\mathbf{Z}$ *and* $\overline{\mathbf{Z}}$.
(b) Let $\neg\mathbf{Z}$ *denote* $(\neg z_1, \ldots \neg z_n)$. *Then* $F(\mathbf{Z}, \mathbf{Y}) \Leftrightarrow \widehat{F}(\mathbf{Z}, \neg\mathbf{Z}, \mathbf{Y})$.

For every $i \in \{1, \ldots n\}$, we can split $\mathbf{Z}$ in two parts, $\mathbf{Z}_1^i$ and $\mathbf{Z}_{i+1}^n$ (assume $\mathbf{Z}_{i+1}^n$ to be the empty sequence if $i = n$), and represent $\widehat{F}(\mathbf{Z}, \overline{\mathbf{Z}}, \mathbf{Y})$ as $\widehat{F}(\mathbf{Z}_1^i, \mathbf{Z}_{i+1}^n, \overline{\mathbf{Z}}_1^i, \overline{\mathbf{Z}}_{i+1}^n, \mathbf{Y})$. We use these representations of $\widehat{F}$ interchangeably, depending on the context. For $b, c \in \{0, 1\}$, let $\mathbf{b}^i$ (resp. $\mathbf{c}^i$) denote a vector of i b's (resp. c's). For notational convenience, we use $\widehat{F}(\mathbf{b}^i, \mathbf{Z}_{i+1}^n, \mathbf{c}^i, \overline{\mathbf{Z}}_{i+1}^n, \mathbf{Y})$ to denote $\widehat{F}(\mathbf{Z}_1^i, \mathbf{Z}_{i+1}^n, \overline{\mathbf{Z}}_1^i, \overline{\mathbf{Z}}_{i+1}^n, \mathbf{Y})|_{\mathbf{Z}_1^i = \mathbf{b}^i, \overline{\mathbf{Z}}_1^i = \mathbf{c}^i}$ in the subsequent discussion. The following is an easy consequence of Proposition 2.

Proposition 3 *For every* $i \in \{1, \ldots n\}$, *the following holds:*
$\widehat{F}(\mathbf{0}^i, \mathbf{Z}_{i+1}^n, \mathbf{0}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y}) \;\Rightarrow\; \exists \mathbf{Z}_1^i F(\mathbf{Z}, \mathbf{Y}) \;\Rightarrow\; \widehat{F}(\mathbf{1}^i, \mathbf{Z}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y})$

Example 2 Consider the specification $F(\mathbf{Z}, \mathbf{Y})$ in Example 1. It is an easy exercise to show that

$$\begin{array}{rcl}
\exists \mathbf{Z}_1^1 F(\mathbf{Z}, \mathbf{Y}) & = & (y_1 \vee \neg z_2) \wedge (z_2 \vee \neg y_2) \wedge (\neg z_2 \vee \neg z_3 \vee \neg y_1) \wedge (z_3 \vee y_1) \wedge (\neg z_3 \vee y_2) \\
\exists \mathbf{Z}_1^2 F(\mathbf{Z}, \mathbf{Y}) & = & \big((y_1 \wedge \neg z_3) \vee \neg y_2\big) \wedge (z_3 \vee y_1) \wedge (\neg z_3 \vee y_2) \\
\exists \mathbf{Z}_1^3 F(\mathbf{Z}, \mathbf{Y}) & = & y_1
\end{array}$$

In addition, we have

$$\begin{array}{rcl}
\widehat{F}(\mathbf{0}^1, \mathbf{Z}_2^3, \mathbf{0}^1, \neg\mathbf{Z}_2^3, \mathbf{Y}) & = & y_1 \wedge \neg z_2 \wedge \neg y_2 \wedge \neg z_3 \\
\widehat{F}(\mathbf{0}^2, \mathbf{Z}_3^3, \mathbf{0}^2, \neg\mathbf{Z}_3^3, \mathbf{Y}) & = & 0 \\
\widehat{F}(\mathbf{0}^3, \mathbf{0}^3, \mathbf{Y}) & = & 0 \\
\widehat{F}(\mathbf{1}^1, \mathbf{Z}_2^3, \mathbf{1}^1, \neg\mathbf{Z}_2^3, \mathbf{Y}) & = & (z_2 \vee \neg y_2) \wedge (\neg z_2 \vee \neg z_3 \vee \neg y_1) \wedge (z_3 \vee y_1) \wedge (\neg z_3 \vee y_2) \\
\widehat{F}(\mathbf{1}^2, \mathbf{Z}_3^3, \mathbf{1}^2, \neg\mathbf{Z}_3^3, \mathbf{Y}) & = & (z_3 \vee y_1) \wedge (\neg z_3 \vee y_2) \\
\widehat{F}(\mathbf{1}^3, \mathbf{1}^3, \mathbf{Y}) & = & 1
\end{array}$$

Notice that $\widehat{F}(\mathbf{0}^i, \mathbf{Z}_{i+1}^n, \mathbf{0}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y}) \Rightarrow \exists \mathbf{Z}_1^i F(\mathbf{Z}, \mathbf{Y}) \Rightarrow \widehat{F}(\mathbf{1}^i, \mathbf{Z}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y})$ holds for each $i \in \{1, 2, 3\}$. □

Lemma 1 *For every $z_i \in \mathbf{Z}$, we have:*

(a) $\neg\widehat{F}(\mathbf{1}^{i-1}0, \mathbf{Z}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y}) \Rightarrow \Delta_i \Rightarrow \neg\widehat{F}(\mathbf{0}^i, \mathbf{Z}_{i+1}^n, \mathbf{0}^{i-1}1, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y})$

(b) $\neg\widehat{F}(\mathbf{1}^i, \mathbf{Z}_{i+1}^n, \mathbf{1}^{i-1}0, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y}) \Rightarrow \Gamma_i \Rightarrow \neg\widehat{F}(\mathbf{0}^{i-1}1, \mathbf{Z}_{i+1}^n, \mathbf{0}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y})$

Proof Follows immediately from proposition 3 and the definitions of Δ_i and Γ_i. □

Example 3 Consider the specification in Example 1 again. The following are easily obtained from the definitions of Δ_i and Γ_i, and from the formulas derived in Example 2.

- $\neg\widehat{F}(0, \mathbf{Z}_2^3, 1, \neg\mathbf{Z}_2^3, \mathbf{Y}) \Leftrightarrow \Delta_1 \Leftrightarrow \neg y_1 \vee (\neg z_2 \wedge y_2) \vee (z_2 \wedge z_3) \vee (z_3 \vee \neg y_2)$
- $\neg\widehat{F}(1, \mathbf{Z}_2^3, 0, \neg\mathbf{Z}_2^3, \mathbf{Y}) \Leftrightarrow \Gamma_1 \Leftrightarrow z_2 \vee y_2 \vee \neg y_1 \vee z_3$
- $\neg\widehat{F}(\mathbf{1}^1, 0, \mathbf{Z}_3^3, \mathbf{1}^1, 1, \neg\mathbf{Z}_3^3, \mathbf{Y}) \Leftrightarrow \Delta_2 \Leftrightarrow \neg\widehat{F}(\mathbf{0}^1, 0, \mathbf{Z}_3^3, \mathbf{0}^1, 1, \neg\mathbf{Z}_3^3, \mathbf{Y}) \Leftrightarrow \neg y_1 \vee y_2 \vee z_3$
- $\neg\widehat{F}(\mathbf{1}^1, 1, \mathbf{Z}_3^3, \mathbf{1}^1, 0, \neg\mathbf{Z}_3^3, \mathbf{Y}) \Leftrightarrow (z_3 \wedge y_1) \vee (\neg z_3 \wedge \neg y_1) \vee (z_3 \wedge \neg y_2) \Rightarrow \neg y_1 \vee z_3 \Leftrightarrow \Gamma_2 \Rightarrow \neg\widehat{F}(\mathbf{0}^1, 1, \mathbf{Z}_3^3, \mathbf{0}^1, 0, \neg\mathbf{Z}_3^3, \mathbf{Y}) \Leftrightarrow 1$
- $\neg\widehat{F}(\mathbf{1}^2, 0, \mathbf{1}^2, 1, \mathbf{Y}) \Leftrightarrow \Delta_3 \Leftrightarrow \neg y_1 \Rightarrow 1 \Leftrightarrow \neg\widehat{F}(\mathbf{0}^2, 0, \mathbf{0}^2, 1, \mathbf{Y})$
- $\neg\widehat{F}(\mathbf{1}^2, 1, \mathbf{1}^2, 0, \mathbf{Y}) \Leftrightarrow \neg y_2 \Rightarrow 1 \Leftrightarrow \Gamma_3 \Leftrightarrow \neg\widehat{F}(\mathbf{0}^2, 1, \mathbf{0}^2, 0, \mathbf{Y})$

As can be seen, in the context of this example, some of the implications in Lemma 1 are strict (i.e. one-way implications), while others are equivalences (i.e. two-way implications). □

Since Δ_i and Γ_i are hard to compute exactly, we mostly use their under-approximations in the development of our synthesis algorithms. Recall from Section 2 that both Δ_i and $\neg\Gamma_i$ suffice as Skolem functions for x_i. Therefore, we propose to use either an under-approximation of Δ_i or an over-approximation of $\neg\Gamma_i$ (depending on which has a smaller AIG) as our approximation of ψ_i. Specifically, we use

$$\delta_i = \neg\widehat{F}(\mathbf{1}^{i-1}0, \mathbf{Z}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y}),\ \gamma_i = \neg\widehat{F}(\mathbf{1}^i, \mathbf{Z}_{i+1}^n, \mathbf{1}^{i-1}0, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y}) \quad (3)$$

$$\psi_i = \delta_i \text{ or } \neg\gamma_i, \text{ depending on which has a smaller AIG} \quad (4)$$

Note that if ψ_i is chosen as δ_i, it under-approximates a correct Skolem function, while if ψ_i is chosen as $\neg\gamma_i$, it over-approximates a correct Skolem function.

Example 4 Consider the specification $\mathbf{Z} = \mathbf{Y}$, expressed in NNF as $F(\mathbf{Z}, \mathbf{Y}) \equiv \bigwedge_{i=1}^n ((z_i \wedge y_i) \vee (\neg z_i \wedge \neg y_i))$. As noted in [38], this is a difficult example for CEGAR-based QBF solvers, when n is large.

From Eqn 3, $\delta_i = \neg(\neg y_i \;\wedge\; \bigwedge_{j=i+1}^{n}(z_j \Leftrightarrow y_j)) = y_i \;\vee\; \bigvee_{j=i+1}^{n}(z_j \Leftrightarrow \neg y_j)$, and $\gamma_i = \neg(y_i \;\wedge\; \bigwedge_{j=i+1}^{n}(z_j \Leftrightarrow y_j)) = \neg y_i \;\vee\; \bigvee_{j=i+1}^{n}(z_j \Leftrightarrow \neg y_j)$. With δ_i as the choice of ψ_i, we obtain $\psi_i = y_i \;\vee\; \bigvee_{j=i+1}^{n}(z_j \Leftrightarrow \neg y_j)$. Clearly, $\psi_n = y_n$. On reverse-substituting, we get $\psi_{n-1} = y_{n-1} \;\vee\; (\psi_n \Leftrightarrow \neg y_n) = y_{n-1} \vee\; 0 = y_{n-1}$. Continuing in this way, we get $\psi_i = y_i$ for all $i \in \{1, \ldots n\}$. The same result is obtained regardless of whether we choose δ_i or $\neg\gamma_i$ for each ψ_i. Thus, our approximation is good enough to solve this problem. In fact, it can be shown that $\delta_i = \Delta_i$ and $\gamma_i = \Gamma_i$ for all $i \in \{1, \ldots n\}$ in this example. □

Note that the approximations of Skolem functions, as given in Equations (3) and (4), are efficiently computable for all $i \in \{1, \ldots n\}$, as they involve evaluating $\widehat{F}$ with a subset of inputs set to constants. This takes no more than $\mathcal{O}(|F|)$ time and space. As illustrated by Example 4, these approximations also often suffice to solve BFnS. The following theorem partially explains this.

Theorem 2 *(a) Suppose* $1 \leq i \leq n$ *and the following holds:*

$$\forall j \in \{1, \ldots i\} \;\; \widehat{F}(\mathbf{1}^j, \mathbf{Z}_{j+1}^n, \mathbf{1}^j, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y}) \Rightarrow \quad \widehat{F}(\mathbf{1}^{j-1}1, \mathbf{Z}_{j+1}^n, \mathbf{1}^{j-1}0, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y}) \vee \; \widehat{F}(\mathbf{1}^{j-1}0, \mathbf{Z}_{j+1}^n, \mathbf{1}^{j-1}1, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y})$$

Then $\exists \mathbf{Z}_1^i F(\mathbf{Z}, \mathbf{Y}) \Leftrightarrow \widehat{F}(\mathbf{1}^i, \mathbf{Z}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y})$.
(b) If $\widehat{F}(\mathbf{Z}, \neg\mathbf{Z}, \mathbf{Y})$ *is in* wDNNF*, then* $\delta_i = \Delta_i$ *and* $\gamma_i = \Gamma_i$ *for every* i *in* $\{1, \ldots n\}$.

Proof To prove part (a), we use induction on i. The base case corresponds to $i = 1$. Recall that $\exists \mathbf{Z}_1^1 F(\mathbf{Z}, \mathbf{Y}) \Leftrightarrow \widehat{F}(1, \mathbf{Z}_2^n, 0, \neg\mathbf{Z}_2^n, \mathbf{Y}) \vee F(0, \mathbf{Z}_2^n, 1, \neg\mathbf{Z}_2^n, \mathbf{Y})$ by definition. Proposition 3 already asserts that $\exists \mathbf{Z}_1^1 F(\mathbf{Z}, \mathbf{Y}) \Rightarrow \widehat{F}(1, \mathbf{Z}_2^n, 1, \neg\mathbf{Z}_2^n, \mathbf{Y})$. Therefore, if the condition in Theorem 2(a) holds for $i = 1$, we have $\widehat{F}(1, \mathbf{Z}_2^n, 1, \neg\mathbf{Z}_2^n, \mathbf{Y}) \Leftrightarrow \widehat{F}(1, \mathbf{Z}_2^n, 0, \neg\mathbf{Z}_2^n, \mathbf{Y}) \vee F(0, \mathbf{Z}_2^n, 1, \neg\mathbf{Z}_2^n, \mathbf{Y})$, which in turn is equivalent to $\exists \mathbf{Z}_1^1 F(\mathbf{Z}, \mathbf{Y})$. This proves the base case.

Let us now assume (inductive hypothesis) that the statement of Theorem 2(a) holds for $1 \leq i < n$. We prove below that the same statement holds for $i+1$ as well. Clearly, $\exists \mathbf{Z}_1^{i+1} F(\mathbf{Z}, \mathbf{Y}) \Leftrightarrow \exists z_{i+1} \left(\exists \mathbf{Z}_1^i F(\mathbf{Z}, \mathbf{Y})\right)$. By the inductive hypothesis, this is equivalent to $\exists z_{i+1} \widehat{F}(\mathbf{1}^i, \mathbf{Z}_{i+1}^n, \mathbf{1}^i, \neg\mathbf{Z}_{i+1}^n, \mathbf{Y})$. By definition of existential quantification, this is equivalent to $\widehat{F}(\mathbf{1}^{i+1}, \mathbf{Z}_{i+2}^n, \mathbf{1}^i 0, \neg\mathbf{Z}_{i+2}^n, \mathbf{Y}) \vee \widehat{F}(\mathbf{1}^i 0, \mathbf{Z}_{i+2}^n, \mathbf{1}^{i+1}, \neg\mathbf{Z}_{i+2}^n, \mathbf{Y})$. From the condition in Theorem 2(a), we also have $\widehat{F}(\mathbf{1}^{i+1}, \mathbf{Z}_{i+2}^n, \mathbf{1}^{i+1}, \overline{\mathbf{Z}}_{i+2}^n, \mathbf{Y}) \Rightarrow \widehat{F}(\mathbf{1}^{i+1}, \mathbf{Z}_{i+2}^n, \mathbf{1}^i 0, \overline{\mathbf{Z}}_{i+2}^n, \mathbf{Y}) \vee \widehat{F}(\mathbf{1}^i 0, \mathbf{Z}_{i+2}^n, \mathbf{1}^{i+1}, \overline{\mathbf{Z}}_{i+2}^n, \mathbf{Y})$. The implication in the reverse direction follows from Proposition 2(a). Thus we have a bi-implication above, which we have already seen is equivalent to $\exists \mathbf{Z}_1^{i+1} F(\mathbf{Z}, \mathbf{Y})$. This proves the inductive case.

To prove part (b), we first show that if $\widehat{F}(\mathbf{Z}, \neg\mathbf{Z}, \mathbf{Y})$ is in wDNNF, then the condition in Theorem 2(a) must hold for all $j \in \{1, \ldots n\}$. Theorem 2(b) then follows from the definitions of Δ_i and Γ_i (see Section 2), from the statement of Theorem 2(a) and from the definitions of δ_i and γ_i (see Eqn 3).

For $1 \leq j \leq n$, let $\zeta(\mathbf{Z}_{j+1}^n, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y})$ denote the negation of the implication in the condition of Theorem 2(a), i.e. $\zeta(\mathbf{Z}_{j+1}^n, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y}) \equiv \widehat{F}(\mathbf{1}^j, \mathbf{Z}_{j+1}^n, \mathbf{1}^j, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y}) \wedge \neg\left(\widehat{F}(\mathbf{1}^{j-1}1, \mathbf{Z}_{j+1}^n, \mathbf{1}^{j-1}0, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y}) \vee \widehat{F}(\mathbf{1}^{j-1}0, \mathbf{Z}_{j+1}^n, \mathbf{1}^{j-1}1, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y})\right)$. To prove by contradiction, suppose $\widehat{F}(\mathbf{Z}, \neg\mathbf{Z}, \mathbf{Y})$ is in wDNNF but there exists j $(1 \leq j \leq n)$ such that $\zeta(\mathbf{Z}_{j+1}^n, \overline{\mathbf{Z}}_{j+1}^n, \mathbf{Y})$ is satisfiable. Let $\mathbf{Z}_{j+1}^n = \sigma$, $\overline{\mathbf{Z}}_{j+1}^n = \kappa$ and $\mathbf{Y} = \theta$ be a satisfying assignment of ζ. We now consider the simplified DAG (circuit) obtained by substituting $\mathbf{1}^{j-1}$ for $\mathbf{Z}_1^{j-1}$ as well as for $\overline{\mathbf{Z}}_1^{j-1}$, σ for $\mathbf{Z}_{j+1}^n$, κ for $\overline{\mathbf{Z}}_{j+1}^n$ and θ for $\mathbf{Y}$ in the DAG representation of $\widehat{F}$. This simplification replaces the output of every internal node with a constant (0 or 1), if the node evaluates to a constant under the above assignment. Note that the resulting DAG (circuit) can have only z_j and $\overline{z}_j$ as its leaves (inputs). Furthermore, since the assignment satisfies ζ, it follows that the simplified circuit evaluates to 1 if both z_j and $\overline{z}_j$ are set to 1, and it evaluates to 0 if any one of z_j or $\overline{z}_j$ is set to 0. This can only happen if there is a node labeled $\wedge$ in the DAG representing $\widehat{F}(\mathbf{Z}, \neg\mathbf{Z}, \mathbf{Y})$ with a path leading to the leaf labeled z_j, and another path leading to the leaf labeled $\overline{z}_j$. This contradicts the assumption that $\widehat{F}(\mathbf{Z}, \neg\mathbf{Z}, \mathbf{Y})$ is in wDNNF. Therefore, there is no $j \in \{1, \ldots n\}$ such that the condition of Theorem 2(a) is violated. $\square$

In general, the candidate Skolem functions generated from the approximations discussed above may not always be correct. Indeed, the conditions discussed above are only sufficient, but not necessary, for the approximations to be exact. Hence, we need a separate check to see if our candidate Skolem function vector $\mathbf{\Psi}$ is correct. To do this, we construct an *error formula* $\varepsilon_{\mathbf{\Psi}}(\mathbf{Z}', \mathbf{Z}, \mathbf{Y}) \equiv F(\mathbf{Z}', \mathbf{Y}) \wedge \bigwedge_{i=1}^n (z_i \Leftrightarrow \psi_i) \wedge \neg F(\mathbf{Z}, \mathbf{Y})$, as described in [28], and check its satisfiability. The first term in the error formula checks if there exists some valuation of $\mathbf{Z}$ that makes $F(\mathbf{Z}, \mathbf{Y})$ true. The second term assigns variables in $\mathbf{Z}$ to the values given by the candidate Skolem functions, and the third term checks if this assignment falsifies the formula F. As shown in [28], checking the unsatisfiability of $\varepsilon_{\mathbf{\Psi}}$ suffices to determine if $\mathbf{\Psi}$ is a correct Skolem function vector. We reproduce below the relevant theorem and proof from [28] for the sake of completeness.

Theorem 3 *$\varepsilon_{\mathbf{\Psi}}$ is unsatisfiable iff $\mathbf{\Psi}$ is a Skolem function vector.*

Proof Suppose $\varepsilon_{\mathbf{\Psi}}$ is unsatisfiable. By definition of $\varepsilon_{\mathbf{\Psi}}$, we have

$$\forall \mathbf{Z}' \forall \mathbf{Z} \forall \mathbf{Y} \left(F(\mathbf{Z}', \mathbf{Y}) \Rightarrow \left(\bigwedge_{i=1}^{n} (z_i \Leftrightarrow \psi_i) \quad\Rightarrow\quad F(\mathbf{Z}, \mathbf{Y}) \right) \right).$$

By standard logic transformations, this implies $\forall \mathbf{Y} \left(\exists \mathbf{Z}' F(\mathbf{Z}', \mathbf{Y}) \;\Rightarrow\; F'(\mathbf{Y}) \right)$, where $F'(\mathbf{Y})$ denotes $F(\mathbf{Z}, \mathbf{Y})$ with z_i substituted by ψ_i for all i in $\{1, \ldots n\}$. Therefore, $\mathbf{\Psi}$ is a Skolem function vector for $\mathbf{Z}$ in F.

Suppose π is a satisfying assignment of $\varepsilon_{\mathbf{\Psi}}$. By definition of $\varepsilon_{\mathbf{\Psi}}$, π is a satisfying assignment of $F(\mathbf{Z}', \mathbf{Y})$ and of $\bigwedge_{i=1}^n (z_i \Leftrightarrow \psi_i) \wedge \neg F(\mathbf{Z}, \mathbf{Y})$, considered separately. Thus, the values of $z_1, \ldots, z_n$ given by $\psi_1, \ldots, \psi_n$ respectively, cause F to evaluate to 0 for the valuation of $\mathbf{Y}$ in π. However, there exists a valuation of $\mathbf{Z}$, viz. $\pi{\downarrow}_{\mathbf{Z}'}$, that causes F to evaluate to 1 for the same valuation of $\mathbf{Y}$ in π. Hence, $\mathbf{\Psi}$ is not a Skolem function vector for $\mathbf{Z}$ in F, as witnessed by the valuation of $\mathbf{Y}$ in π. $\square$

We now combine all the above ingredients to come up with algorithm BFSS (for *Blazingly Fast Skolem Synthesis*), as shown in Algorithm 1. The algorithm can be divided into three parts. In the first part (lines 2-10), unateness is checked. This is done in two ways: (i) we identify pure literals in F by simply examining the labels of leaves in the DAG representation of F in NNF, and (ii) we check the satisfiability of the formulas η_i^+ and η_i^-, as defined in Equations (1) and (2). This requires invoking a SAT solver in the worst-case, and the solver may need to be

Algorithm 1: BFSS

Input: $F(\mathbf{Z},\mathbf{Y})$ in NNF with inputs $\mathbf{Y}$ and outputs $\mathbf{Z}$. Let $|\mathbf{Y}| = m$ and $|\mathbf{Z}| = n$
Output: Skolem function vector $\mathbf{\Psi} = (\psi_1, \ldots, \psi_n)$ for $\mathbf{Z}$ in F

Initialize: $U_0 := \emptyset$; $U_1 := \emptyset$; `// Sets of negative and positive unate variables`
repeat
  for *each* $z_i \in \mathbf{Z} \setminus (U_0 \cup U_1)$ **do**
    if *F is positive unate in z_i* `//` z_i `pure or` η_i^+ `(Eqn 1) satisfiable` ;
    then
      $F := F[z_i = 1]$; $U_1 := U_1 \cup \{z_i\}$;
    else if *F is negative unate in z_i* `//` $\neg z_i$ `pure or` η^- `(Eqn 2) satisfiable` ;
    then
      $F := F[z_i = 0]$; $U_0 := U_0 \cup \{z_i\}$;
until *no more unate variables found*;
Choose an ordering $\preceq$ of $\mathbf{Z}$; `// Section 6 discusses actual ordering used;`
for *each* $z_i \in \mathbf{Z}$ *in* $\preceq$ *order* **do**
  if *$z_i \in U_j$ for $j \in \{0,1\}$* `// Assume` $z_1 \preceq z_2 \preceq \ldots z_n$;
  then
    $\psi_i := j$;
  else
    Compute δ_i, γ_i and ψ_i according to Equations (3) and (4);
$\varepsilon_{\mathbf{\Psi}} := F(\mathbf{Z}', \mathbf{Y}) \wedge \bigwedge_{i=1}^{n}(z_i \Leftrightarrow \psi_i) \wedge \neg F(\mathbf{Z},\mathbf{Y})$;
if *$\varepsilon_{\mathbf{\Psi}}$ is unsatisfiable* **then**
  Terminate and output $\mathbf{\Psi}$;
else
  Call Phase2;

invoked at most $\mathcal{O}(n^2)$ times until no more unate variables are detected. Once we have done this, by Proposition 1, the constants 1 and 0 are correct Skolem functions for the positive and negative unate variables respectively, thus identified.

In the second part, we fix an ordering of the remaining output variables according to an experimentally sound heuristic, as described in Section 6, and compute candidate Skolem functions for these variables according to Equations (3) and (4). We then check the satisfiability of the error formula $\epsilon_{\mathbf{\Psi}}$ to determine if the candidate Skolem functions are indeed correct. If the error formula is found to be unsatisfiable, we know from Theorem 3 that we have correct Skolem functions, which can therefore be output. This concludes phase 1 of algorithm BFSS. However, if the error formula is found to be satisfiable, we move to phase 2 of algorithm BFSS. It is not difficult to see that the running time of phase 1 is polynomial in the size of the input, relative to an NP-oracle (SAT solver in practice). This also implies that the Skolem functions generated can be of at most polynomial size. Finally, if F satisfies the conditions of Theorem 2, the Skolem functions generated in phase 1 are correct. From the above reasoning, we obtain the following properties of phase 1 of BFSS:

Theorem 4 *1. For all output variables in which F is unate, phase 1 of* BFSS *computes correct Skolem functions.*
2. If $\hat{F}$ is in wDNNF*, phase 1 of* BFSS *computes correct Skolem functions.*
3. The running time of phase 1 of BFSS *is polynomial in input size, relative to an* NP*-oracle. Specifically, the algorithm makes $\mathcal{O}(n^2)$ calls to an* NP*-oracle.*

4. The candidate Skolem functions output by phase 1 of BFSS *have size at most polynomial in the size of the input.*

By our hardness results in Section 3, we know that the above algorithm cannot solve BFnS for all inputs, unless some well-regarded complexity-theoretic conjectures fail. As a result, we must go to phase 2, in the worst case. Our experiments however show that this is not necessary in the majority of the benchmarks and phase 1 itself suffices. Interestingly, this is despite the fact that not all of the benchmarks are in wDNNF. Indeed, there is a deeper connection between the representation of the specification F and the complexity of synthesis of Skolem functions, as has been explored recently in [2].

5 Synthesis by expansion

We now describe phase 2 of BFSS, which is invoked only if phase 1 fails to generate a correct Skolem function vector. Unlike phase 1, phase 2 may need exponentially many invocations of an NP-oracle in the worst case. However, phase 2 always terminates with a correct Skolem function vector.

Recall that the candidate Skolem functions computed in Step 17 of Algorithm 1 were derived from under-approximations δ_i and γ_i of Δ_i and Γ_i respectively. As discussed in Section 2, if we could use Δ_i and Γ_i instead, we would obtain the correct Skolem functions directly. This suggests a generic method for "improving" the candidate Skolem functions obtained from phase 1. Specifically, we propose to *expand* the under-approximations δ_i and/or γ_i, while maintaining the invariant $(\delta_i \Rightarrow \Delta_i) \wedge (\gamma_i \Rightarrow \Gamma_i)$ for all $i \in \{1, \ldots n\}$. Formally, we say δ_i' is an *expansion* of δ_i if $(\delta_i \Rightarrow \delta_i' \Rightarrow \Delta_i) \wedge (\delta_i' \not\Rightarrow \delta_i)$ holds. Similarly, we say γ_i' is an expansion of γ_i if $(\gamma_i \Rightarrow \gamma_i' \Rightarrow \Gamma_i) \wedge (\gamma_i' \not\Rightarrow \gamma_i)$ holds. Note that the candidate Skolem function δ_i' (resp. $\neg\gamma_i'$) is "better" than δ_i (resp. $\neg\gamma_i$) in the sense that it differs from the correct Skolem function Δ_i (resp. $\neg\Gamma_i$) on strictly fewer assignments.. In the limit, if δ_i (resp. γ_i) is expanded all the way to be semantically equivalent to Δ_i (resp. Γ_i), the candidate Skolem function ψ_i is indeed a correct Skolem function.

In general, different algorithms may be used for expanding δ_i and/or γ_i, i.e. obtaining δ_i' and/or γ_i' satisfying the expansion conditions given above. We use the term *expansion-based algorithm* to denote any algorithm for Boolean functional synthesis that works by starting with underapproximations of Δ_i and/or Γ_i for every output z_i, and that (progressively or in a single step) expands these underapproximations until correct Skolem functions are obained either as δ_i or $\neg\gamma_i$, as the case may be. The counterexample-guided abstraction refinement (CEGAR) algorithm of [28] is a special case of an expansion-based algorithm that works for factored specifications. In phase 2 of BFSS, we use a mix of three different expansion-based algorithms that work for arbitrary specifications.

5.1 Zooming down on a Skolem function to rectify

Suppose $\mathbf{\Psi}$ is a candidate Skolem function vector, where each ψ_i is either δ_i or $\neg\gamma_i$, with $\delta_i \Rightarrow \Delta_i$ and $\gamma_i \Rightarrow \Gamma_i$. Suppose further that π is a satisfying assignment of the error formula $\varepsilon_{\mathbf{\Psi}}$. By Theorem 3, at least one candidate Skolem function ψ_i is incorrect and must be rectified. We call $\pi{\downarrow}_{\mathbf{Y}}$ a *counterexample* for $\mathbf{\Psi}$, since $\mathbf{\Psi}$ fails to serve as a correct Skolem function vector when $\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}}$. Furthermore, since $F(\pi{\downarrow}_{\mathbf{Z}}, \pi{\downarrow}_{\mathbf{Y}}) = 0$, we say that $\pi{\downarrow}_{\mathbf{Z}}$ is the *evidence* for $\pi{\downarrow}_{\mathbf{Y}}$ being a counterexample. Our goal now is to expand δ_i and γ_i, as needed, to ensure that $\pi{\downarrow}_{\mathbf{Y}}$ eventually ceases to be a counterexample. We call this process *eliminating a counterexample*.

Since some Skolem functions in $\mathbf{\Psi}$ may indeed be correct, we must first identify candidate Skolem functions ψ_i that are necessarily incorrect. Recall from Section 2 that for every $i \in \{1, \ldots n\}$, ψ_i is expressed as a function of $z_{i+1}, \ldots z_n$ and $\mathbf{Y}$. Hence, given a candidate Skolem function vector $\mathbf{\Psi}$ and an assignment $\tau : \mathbf{Y} \to \{0, 1\}$, the value of z_n (given by ψ_n) depends only on τ, the value z_{n-1} (given by ψ_{n-1}) depends on the value of z_n (given by ψ_n) and on τ, and so on until z_1. Therefore, if a candidate Skolem function ψ_i is incorrect, it can induce another candidate Skolem function ψ_j to compute an incorrect value for z_j, where $j < i$. In view of this, when finding erroneous candidate Skolem functions, it is desirable that we first examine ψ_n, and only if ψ_n is correct, should we examine ψ_{n-1}, and so on. Hence, finding the largest $k \in \{1, \ldots n\}$ such that ψ_k is incorrect is important when rectifying erroneous candidate Skolem functions. In general, this requires taking into account all counterexamples for $\mathbf{\Psi}$. Since the count of such counterexamples can be exponential in $|\mathbf{Y}|$, we focus for now on the specific counterexample $\pi{\downarrow}_{\mathbf{Y}}$, and find the largest k such that ψ_k is incorrect when $\mathbf{Y}$ is set to $\pi{\downarrow}_{\mathbf{Y}}$. As we show later, rectifying the corresponding ψ_k is not wasted effort, since it *must* be rectified by *every* expansion-based algorithm before a correct Skolem function vector is obtained.

To reduce notational clutter in the following discussion, for every assignment $\tau \in \{0, 1\}^n$ of $\mathbf{Y}$, we use $\mathbf{\Psi}(\tau)$ to denote the sequence $(\xi_1, \ldots \xi_n)$, where $\xi_n = \psi_n(\tau)$ and $\xi_i = \psi_i(\xi_{i+1}, \ldots \xi_n, \tau)$ for $i \in \{1, \ldots n-1\}$. With abuse of notation, we also use $\psi_i(\tau)$ to denote $\psi_i(\xi_{i+1}, \ldots \xi_n, \tau)$ for $i \in \{1, \ldots n-1\}$, when there is no confusion.

Definition 3 Let $\mathbf{\Psi}$ be a candidate Skolem function vector for a specification $F(\mathbf{Z}, \mathbf{Y})$. Let $\tau \in \{0, 1\}^n$ be an assignment of $\mathbf{Y}$ such that $\exists \mathbf{Z}\, F(\mathbf{Z}, \tau) = 1$. We define the *critical index* of $\mathbf{\Psi}$ with respect to τ, denoted $\kappa_{\mathbf{\Psi}}(\tau)$, as follows:

$$\kappa_{\mathbf{\Psi}}(\tau) = 0 \text{ if } F(\mathbf{\Psi}(\tau), \tau) = 1, \text{ and}$$
$$\kappa_{\mathbf{\Psi}}(\tau) = \min_k \big(\exists z_1, \ldots z_k\, F\big(z_1, \ldots z_k, \psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1\big) \text{ otherwise.}$$

Let $k = \kappa_{\mathbf{\Psi}}(\tau)$. Intuitively, if we assign $(\psi_{k+1}(\tau), \ldots \psi_n(\tau))$ to $\mathbf{Z}_{k+1}^n$ and τ to $\mathbf{Y}$, it is possible to satisfy $F(\mathbf{Z}, \mathbf{Y})$ by choosing some values in $\{0, 1\}$ for each of $z_1, \ldots z_k$. However, if we additionally assign $\psi_k(\tau)$ to z_k, there is no way to satisfy $F(\mathbf{Z}, \mathbf{Y})$. Therefore, k is the largest index in $\{1, \ldots n\}$ such that ψ_k is an incorrect candidate Skolem function, when considering the counterexample τ.

Example 5 Let us re-visit the specification from Example 1, reproduced here for convenience: $F(\mathbf{Z}, \mathbf{Y}) = (z_1 \vee y_1) \wedge (\neg z_1 \vee \neg z_2) \wedge (z_2 \vee \neg y_2) \wedge (\neg z_2 \vee \neg z_3 \vee \neg y_1) \wedge (z_3 \vee y_1) \wedge (\neg z_3 \vee y_2)$. Following Equations (3) and (4) and using $\neg\gamma_i$ as the initial candidate Skolem function for z_i, we get $\psi_1 = \neg z_2 \wedge \neg y_2 \wedge y_1 \wedge \neg z_3$, $\psi_2 = (\neg z_3 \vee \neg y_1) \wedge (z_3 \vee y_1) \wedge (\neg z_3 \vee y_2)$ and $\psi_3 = y_2$. The corresponding error formula $\varepsilon_{\mathbf{\Psi}}$ has a satisfying assignment $(z_1', z_2', z_3', z_1, z_2, z_3, y_1, y_2) = (0, 1, 0, 0, 0, 1, 1, 1)$. Hence, $(y_1, y_2) = (1, 1)$ is a counterexample and $(z_1, z_2, z_3) = (0, 0, 1)$ is the evidence for the counterexample. In this case, $F(z_1, z_2, 1, 1, 1) = (\neg z_1 \vee \neg z_2) \wedge z_2 \wedge \neg z_2 = 0$ for all values of z_1, z_2. Hence, ψ_3 is in error, and must be rectified if we are to eliminate the counterexample $(y_1, y_2) = (1, 1)$. Note that by Definition 3, $\kappa_{\mathbf{\Psi}}((1,1))$ equals 3 in this case. □

Recall from Section 4 that the error formula $\varepsilon_{\mathbf{\Psi}}$ has free variables $\mathbf{Z}'$, $\mathbf{Z}$ and $\mathbf{Y}$. Therefore, if π is a satisfying assignment of $\varepsilon_{\mathbf{\Psi}}$, we have $F(\pi{\downarrow_{\mathbf{Z}'}}, \pi{\downarrow_{\mathbf{Y}}}) = 1$ and $F(\pi{\downarrow_{\mathbf{Z}}}, \pi{\downarrow_{\mathbf{Y}}}) = 0$. The following proposition now follows from Definition 3.

Proposition 4 *If $\pi{\downarrow_{\mathbf{Y}}}$ is a counterexample for $\mathbf{\Psi}$, then $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}}) > 0$.*

In the case of Example 5 above, $\mathbf{Y} = (1, 1)$ is a counterexample for $\mathbf{\Psi}$, and indeed $\kappa_{\mathbf{\Psi}}((1,1)) = 3$ (> 0). We now show that regardless of which expansion-based algorithm is used (including those that consider all counterexamples for $\mathbf{\Psi}$), if k denotes $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$, the k^{th} candidate Skolem function *must* be rectified before the counterexample $\pi{\downarrow_{\mathbf{Y}}}$ is eliminated. Towards a formalization of this result, let $\mathcal{A}$ denote an arbitrary expansion-based algorithm that takes $F(\mathbf{Z}, \mathbf{Y})$ and $\mathbf{\Psi}$ as inputs, and returns an updated Skolem function vector $\mathbf{\Psi}'$ as output. The following lemma shows that $\mathbf{\Psi}'$ cannot differ from $\mathbf{\Psi}$ in components with index greater than $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$, if we evaluate them on the counterexample $\pi{\downarrow_{\mathbf{Y}}}$ for $\mathbf{\Psi}$.

Lemma 2 *For all $i \in \{\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}}) + 1, \ldots n\}$, $\psi_i(\pi{\downarrow_{\mathbf{Y}}}) = \psi'_i(\pi{\downarrow_{\mathbf{Y}}})$.*

Proof We prove the lemma by contradiction. For notational convenience, let τ denote $\pi{\downarrow_{\mathbf{Y}}}$ in the proof. If possible, let there be an index $i \in \{\kappa_{\mathbf{\Psi}}(\tau) + 1, \ldots n\}$ such that $\psi_i(\tau) \neq \psi'_i(\tau)$. Without loss of generality, we choose i to be the largest such index. This implies that for all $j \in \{i + 1, \ldots n\}$, $\psi_j(\tau) = \psi'_j(\tau)$.

There are two sub-cases to consider, depending on whether ψ_i was chosen to be δ_i or $\neg\gamma_i$, where $\delta_i \Rightarrow \Delta_i$ and $\gamma_i \Rightarrow \Gamma_i$. We consider the case where ψ_i was chosen to be δ_i first. Since $\psi_i(\tau) \neq \psi'_i(\tau)$, algorithm $\mathcal{A}$ must have changed δ_i. Since $\mathcal{A}$ is an expansion-based algorithm, it can only change δ_i by expanding it. Therefore, we must have $\delta_i(\psi_{i+1}(\tau), \ldots \psi_n(\tau), \tau) = 0$ and $\delta'_i(\psi_{i+1}(\tau), \ldots \psi_n(\tau), \tau) = 1$. This also means that $\psi_i(\tau) = 0$.

Since $\kappa_{\mathbf{\Psi}}(\tau) + 1 \leq i \leq n$, by Definition 3, we have $\exists \mathbf{Z}_1^{i-1}\, F(\mathbf{Z}_1^{i-1}, \psi_i(\tau), \ldots \psi_n(\tau), \tau) = 1$. Furthermore, since $\delta'_i(\psi_{i+1}(\tau), \ldots \psi_n(\tau), \tau) = 1$ and since δ'_i underapproximates Δ_i (recall $\mathcal{A}$ is an expansion-based algorithm), we have $\Delta_i(\psi_{i+1}(\tau), \ldots \psi_n(\tau), \tau) = 1$. Therefore, by definition of Δ_i (see Section 2), $\exists \mathbf{Z}_1^{i-1}\, F(\mathbf{Z}_1^{i-1}, 0, \psi_{i+1}(\tau), \ldots \psi_n(\tau), \tau) = 0$. Since $\psi(\tau) = 0$, this also means $\exists \mathbf{Z}_1^{i-1}\, F(\mathbf{Z}_1^{i-1}, \psi_i(\tau), \ldots \psi_n(\tau), \tau) = 0$ This contradicts what we inferred above. A similar analysis for the sub-case where ψ_i is $\neg\gamma_i$ also leads to a contradiction. This proves the lemma. □

Corollary 1 *Let τ be a counterexample for $\mathbf{\Psi}$, and let $\mathbf{\Psi}'$ be the updated candidate Skolem function vector generated by an expansion-based algorithm $\mathcal{A}$. If $\psi_k(\tau) = \psi'_k(\tau)$, where $k = \kappa_{\mathbf{\Psi}}(\tau)$, then τ is a counterexample for $\mathbf{\Psi}'$ as well.*

Proof From Lemma 2, $\psi_i(\tau) = \psi'_i(\tau)$ for all $i \in \{k + 1, \ldots n\}$. Suppose further that $\psi_k(\tau) = \psi'_k(\tau)$. From the definition of $\kappa_{\mathbf{\Psi}}(\tau)$ (see Definition 3), we know that $\exists \mathbf{Z}_1^{k-1}\, F(\mathbf{Z}_1^{k-1}, \psi_k(\tau), \ldots \psi_n(\tau), \tau) = 0$. It follows that $\exists \mathbf{Z}_1^{k-1}\, F(\mathbf{Z}_1^{k-1}, \psi'_k(\tau), \ldots \psi'_n(\tau), \tau)$ is also 0. Hence, $\neg F(\mathbf{Z}, \tau) \wedge \bigwedge_{i=1}^{n}(z_i \Leftrightarrow \psi'_i(\tau))$ is satisfiable. Furthermore, since τ is a counterexample for $\mathbf{\Psi}$, we know from the definition of $\varepsilon_{\mathbf{\Psi}}$ that $F(\mathbf{Z}', \tau)$ is satisfiable. It follows that $F(\mathbf{Z}', \tau) \wedge \neg F(\mathbf{Z}, \tau) \wedge \bigwedge_{i=1}^{n}(z_i \Leftrightarrow \psi'_i(\tau))$ is satisfiable. In other words, τ is a counterexample for $\mathbf{\Psi}'$. □

Corollary 2 *Once a counterexample is eliminated, it can never be re-introduced by an expansion-based algorithm.*

Proof Let τ be an assignment of $\mathbf{Y}$ that represents an eliminated counterexample. Hence, if $\mathbf{\Psi}$ denotes the current candidate Skolem function vector, we have $F(\mathbf{\Psi}(\tau), \tau) = 1$. By Definition 3, we also have $\kappa_{\mathbf{\Psi}}(\tau) = 0$. Therefore, by Lemma 2, if $\mathbf{\Psi}'$ is the updated candidate Skolem function vector generated by an expansion-based algorithm, we must have $\psi_i'(\tau) = \psi_i(\tau)$ for all $i \in \{1, \ldots n\}$. Hence $F(\mathbf{\Psi}'(\tau), \tau) = F(\mathbf{\Psi}(\tau), \tau) = 1$. Recalling the definition of $\varepsilon_{\mathbf{\Psi}'}$, it follows that τ cannot be a a counterexample for $\mathbf{\Psi}'$. $\square$

Lemma 3 *Let τ be a counterexample for $\mathbf{\Psi}$ with $k = \kappa_{\mathbf{\Psi}}(\tau)$. The following statements are true.*

1. *Any expansion-based algorithm that eliminates the counterexample τ must necessarily update ψ_k.*
2. *If $\psi_k = \delta_k$, then $\delta_k \not\Leftrightarrow \Delta_k$. Specifically, $\delta_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 0$ while $\Delta_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1$.*
3. *If $\psi_k = \neg\gamma_k$, then $\gamma_k \not\Leftrightarrow \Gamma_k$. Specifically, $\gamma_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 0$ while $\Gamma_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1$.*

Proof Part (a) is an easy consequence of Corollary 1. We prove part (b) by contradiction. Suppose, if possible, $\delta_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1$. Since $\delta_k \Rightarrow \Delta_k$, we must have $\Delta_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1$ as well. Thus, both δ_k and Δ_k evaluate to the same value, i.e. 1, for $\mathbf{Z}_{k+1}^n = \big(\psi_{k+1}(\tau), \ldots \psi_n(\tau)\big)$ and $\mathbf{Y} = \tau$. We also know that Δ_k is always a correct Skolem function for z_k. Since ψ_k is chosen as δ_k, it follows that ψ_k evaluates to the value of the correct Skolem function for z_k when $\mathbf{Z}_{k+1}^n = \big(\psi_{k+1}(\tau), \ldots \psi_n(\tau)\big)$ and $\mathbf{Y} = \tau$. Therefore, by the definition of a Skolem function, if $\exists \mathbf{Z}_1^k F\big(\mathbf{Z}_1^k, \psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1$, then $\exists \mathbf{Z}_1^{k-1} F\big(\mathbf{Z}_1^{k-1}, \psi_k(\tau), \ldots \psi_n(\tau), \tau\big) = 1$ as well. However, this contradicts the fact that $k = \kappa_{\mathbf{\Psi}}(\tau)$ (see Definition 3). Therefore, $\delta_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 0$.

To see why $\Delta_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1$, notice that $\exists \mathbf{Z}_1^k F\big(\mathbf{Z}_1^k, \psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big) = 1$, although $\exists \mathbf{Z}_1^{k-1} F\big(\mathbf{Z}_1^{k-1}, \psi_k(\tau), \ldots \psi_n(\tau), \tau\big) = 0$. Therefore, a correct Skolem function for z_k, viz. Δ_k, must evaluate to $\neg\psi_k(\tau)$ when $\mathbf{Z}_{k+1}^n = \big(\psi_k(\tau), \ldots \psi_1(\tau)\big)$ and $\mathbf{Y} = \tau$. We have already seen above that the value of ψ_k $(= \delta_k)$ for this assignment of $\mathbf{Z}_{k+1}^n$ and $\mathbf{Y}$, is 0. In other words, $\psi_k(\tau) = 0$. Therefore, $\Delta_k\big(\psi_{k+1}(\tau), \ldots \psi_n(\tau), \tau\big)$ must evaluate to 1. This also clearly shows that $\delta_k \not\Leftrightarrow \Delta_k$.

The proof for part (c) is exactly the same as that for part (b) with γ_k and Γ_k replacing δ_k and Δ_k, respectively. Since $\psi_k = \neg\gamma_k$ in this case, ψ_k must evaluate to 1, while a correct Skolem function (such as $\neg\Gamma_k$) must evaluate to 0, when $\mathbf{Z}_{k+1}^n = \big(\psi_{k+1}(\tau), \ldots \psi_n(\tau)\big)$ and $\mathbf{Y} = \tau$. $\square$

It is clear from the discussion above that the critical index of $\mathbf{\Psi}$ w.r.t a counterexample $\pi{\downarrow}_{\mathbf{Y}}$ plays an important role in identifying a candidate Skolem function that must be rectified. How do we find this critical index in practice? If π denotes a satisfying assignment of $\varepsilon_{\mathbf{\Psi}}$, it is an easy exercise to show that $\exists \mathbf{Z}_1^i F(\mathbf{Z}_1^i, \pi{\downarrow}_{\mathbf{Z}_{i+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$ logically implies $\exists \mathbf{Z}_1^j F(\mathbf{Z}_1^j, \pi{\downarrow}_{\mathbf{Z}_{j+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$ for all $j \in \{i, \ldots n\}$. Therefore, $\kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$ can be found by a binary search that identifies the minimum i such that $F(\mathbf{Z}_1^i, \pi{\downarrow}_{\mathbf{Z}_{i+1}^n}, \pi{\downarrow}_{\mathbf{Y}})$ is satisfiable. This requires $\mathcal{O}(\log_2 n)$ calls to a SAT solver. In the following discussion, we assume access to a procedure ComputeK that finds $\kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$, given $\mathbf{\Psi}$ and π, in this manner.

5.2 Counterexample-guided expansion of δ_i and γ_i

We now describe three expansion-based algorithms used in phase 2 of BFSS. While we experimented with several expansion-based algorithms, the combination of the three presented below gave us the best results in practice. In the following discussion, we assume that $\mathbf{\Psi}$ is a candidate Skolem function vector, where ψ_i is either δ_i or $\neg\gamma_i$, for each $i \in \{1, \ldots n\}$. Furthermore, we assume that π is a satisfying assignment of $\varepsilon_{\mathbf{\Psi}}$, and $k = \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$. Since $\varepsilon_{\mathbf{\Psi}} = F(\mathbf{Z}', \mathbf{Y}) \wedge \neg F(\mathbf{Z}, \mathbf{Y}) \wedge \bigwedge_{i=1}^{n}(z_i \Leftrightarrow \psi_i)$, it is easy to see that $\pi{\downarrow}_{\mathbf{Z}} = \mathbf{\Psi}\big(\pi{\downarrow}_{\mathbf{Y}}\ \big)$. Therefore, for $1 \leq i \leq j \leq n$, we often use $\pi{\downarrow}_{\mathbf{Z}_i^j}$ to refer to $\big(\psi_i(\pi{\downarrow}_{\mathbf{Y}}), \ldots \psi_j(\pi{\downarrow}_{\mathbf{Y}})\big)$ in the following discussion.

5.2.1 Maximally expanding δ_i and γ_i

In this approach, we make use of the observation that if δ_i and γ_i are maximally expanded to become semantically equivalent to Δ_i and Γ_i respectively, then there is no further need to update the candidate Skolem function ψ_i (chosen to be either δ_i or $\neg\gamma_i$). We know from Definition 3 that there is a satisfying assignment of $F(\mathbf{Z}, \mathbf{Y})$ in which $\mathbf{Z}_{k+1}^n$ has the value $\pi{\downarrow}_{\mathbf{Z}_{k+1}^n}$. Hence, there is no need to update $\psi_{k+1}, \ldots \psi_n$ in order to eliminate the counterexample $\pi{\downarrow}_{\mathbf{Y}}$. Instead, if we simply ensure that all δ_i and γ_i for $i \in \{1, \ldots k\}$ are expanded to Δ_i and Γ_i respectively, the counterexample $\pi{\downarrow}_{\mathbf{Y}}$ is guaranteed to be eliminated. Algorithm MaxExpand (see Algorithm 2) achieves this when the input parameter c is set to k. Note that this algorithm requires $(\delta_1 \Leftrightarrow \Delta_1) \wedge (\gamma_1 \Leftrightarrow \Gamma_1)$ to hold when it is invoked. Fortunately, this pre-condition is trivially satisfied. Specifically, $\Delta_1 = \neg F(0, \mathbf{Z}_2^n, \mathbf{Y})$ by definition, and $\delta_1 = \neg\widehat{F}(0, \mathbf{Z}_2^n, 1, \neg\mathbf{Z}_{2+1}^n, \mathbf{Y}) = \neg F(0, \mathbf{Z}_2^n, \mathbf{Y})$ from Equation (3). It follows that $\delta_1 = \Delta_1$. By a similar argument, we get $\gamma_1 = \Gamma_1$ as well.

Algorithm 2: MaxExpand

```
Input: c ∈ {1, ..., n},  δ_1, γ_1
Output: Updated (δ_i, γ_i, ψ_i) for 1 ≤ i ≤ c
// Requires: (δ_1 ⇔ Δ_1) ∧ (γ_1 ⇔ Γ_1)
for i = 2 to c do
    δ_i ← (δ_{i−1} ∧ γ_{i−1})|_{z_i=0};
    γ_i ← (δ_{i−1} ∧ γ_{i−1})|_{z_i=1};
for i = 1 to c do
    ψ_i ← δ_i (or ¬γ_i);                    // Either choice is fine
return (δ_i, γ_i, ψ_i) for 1 ≤ i ≤ c;
```

Lemma 4 *The following statements hold after Algorithm* MaxExpand *terminates, where primed functions denote their updated versions after executing the algorithm.*

1. *$\delta_i' \Leftrightarrow \Delta_i$ and $\gamma_i' \Leftrightarrow \Gamma_i$ for all $i \in \{1, \ldots c\}$.*
2. *Let $\mathbf{\Psi}'$ be the updated Skolem function vector that results from setting ψ_i' to either δ_i' or $\neg\gamma_i'$ for all $i \in \{1, \ldots n\}$. If $\pi' \models \varepsilon_{\mathbf{\Psi}'}$, then $\kappa_{\mathbf{\Psi}}(\pi'{\downarrow}_{\mathbf{Y}}) > c$.*

Proof We prove part (1) by induction on c. By virtue of the pre-condition, the base case is satisfied when $c = 1$. Suppose the claim holds for all i in $\{1, \ldots m\}$, where $1 \leq m < c$. Thus, $\delta'_m = \Delta_m$ and $\gamma'_m = \Gamma_m$. We now show that the claim holds for $m+1$ as well. By definition, $\Delta_{m+1} = \neg\exists \mathbf{Z}_1^m F(\mathbf{Z}_1^m, 0, \mathbf{Z}_{m+2}^n, \mathbf{Y}) = \neg\left(\exists \mathbf{Z}_1^{m-1} F(\mathbf{Z}_1^{m-1}, 0, \mathbf{Z}_{m+1}^n, \mathbf{Y})|_{z_{m+1}=0} \vee \exists \mathbf{Z}_1^{m-1} F(\mathbf{Z}_1^{m-1}, 1, \mathbf{Z}_{m+1}^n, \mathbf{Y})|_{z_{m+1}=0}\right)$. This, in turn, is equivalent to $(\Delta_m \wedge \Gamma_m)|_{z_{m+1}=0}$. Therefore, using the induction hypothesis, we get $\Delta_{m+1} = \left(\delta'_m \wedge \gamma'_m\right)|_{z_{m+1}=0}$. A similar argument shows that $\Gamma_{m+1} = \left(\delta'_m \wedge \gamma'_m\right)|_{z_{m+1}=1}$. By mathematical induction, and by virtue of the updates in steps 2 and 3 of MaxExpand, we finally get $\delta'_i = \Delta_i$ and $\gamma'_i = \Gamma_i$ for $1 \leq i \leq c$.

To prove part (2), suppose $\pi' \models \varepsilon_{\mathbf{\Psi}'}$ and $l = \kappa_{\mathbf{\Psi}}(\pi'{\downarrow}_{\mathbf{Y}}) \leq c$. By Lemmas 3(2) and 3(3), either $(\delta'_l \not\Leftrightarrow \Delta_l)$ or $(\gamma'_l \not\Leftrightarrow \Gamma_l)$ must hold. This contradicts the first part of the lemma proved above. Hence $\kappa_{\mathbf{\Psi}}(\pi'{\downarrow}_{\mathbf{Y}})$ must be greater than c. $\square$

The worst-case size of the updated δ_i and γ_i functions computed by Algorithm MaxExpand grows exponentially in c and linearly in $|F|$. This blow-up is similar to that seen in the algorithm of Jiang et al. for quantifier elimination via functional composition [25, 7]. Therefore, although Algorithm MaxExpand can solve BFnS in principle (if the parameter c is set to n), it is useful in practice only when c is restricted to small values (say, ≤ 4).

5.2.2 Expanding to reduce the critical index

In this approach, we expand γ_i and/or δ_i in a manner that ensures that the critical index of $\mathbf{\Psi}$ w.r.t. the counterexample $\pi{\downarrow}_{\mathbf{Y}}$ reduces. By Proposition 4, we know that the critical index of $\mathbf{\Psi}$ w.r.t. a counterexample must always be positive. Hence, it can reduce at most n $(= |\mathbf{Z}|)$ times, after which the counterexample must be eliminated.

Since k is the critical index of $\mathbf{\Psi}$ w.r.t. $\pi{\downarrow}_{\mathbf{Y}}$, we know from Definition 3 that $\exists \mathbf{Z}_1^{k-1} F(\mathbf{Z}_1^{k-1}, \pi{\downarrow}_{\mathbf{Z}_k^n}, \pi{\downarrow}_{\mathbf{Y}}) = 0$ and $\exists \mathbf{Z}_1^k F(\mathbf{Z}_1^k, \pi{\downarrow}_{\mathbf{Z}_{k+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$. It follows from elementary logic that $\exists \mathbf{Z}_1^{k-1} F(\mathbf{Z}_1^{k-1}, \neg\pi[z_k], \pi{\downarrow}_{\mathbf{Z}_{k+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$. This fact, together with Lemma 2, suggests that we update the candidate Skolem function ψ_k so that that it evaluates to $\neg\pi[z_k]$ (instead of $\pi[z_k]$, as it currently does) when $\mathbf{Z}_{k+1}^n$ and $\mathbf{Y}$ are set to $\pi{\downarrow}_{\mathbf{Z}_{k+1}^n}$ and $\pi{\downarrow}_{\mathbf{Y}}$ respectively. Let the updated Skolem function vector be $\mathbf{\Psi}'$. Clearly, the critical index of $\mathbf{\Psi}'$ w.r.t. $\pi{\downarrow}_{\mathbf{Y}}$ cannot be k or more, since $\exists \mathbf{Z}_1^{k-1} F(\mathbf{Z}_1^{k-1}, \neg\pi[z_k], \pi{\downarrow}_{\mathbf{Z}_{k+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$. Therefore, either $\pi{\downarrow}_{\mathbf{Y}}$ ceases to be a counterexample, or the critical index of $\mathbf{\Psi}'$ w.r.t. $\pi{\downarrow}_{\mathbf{Y}}$ reduces to a value strictly less than k.

Lemmas 3(2) and 3(3) suggest that if δ_k (resp. γ_k) is updated to evaluate to 1 when $\mathbf{Z}_{k+1}^n$ and $\mathbf{Y}$ are set to $\pi{\downarrow}_{\mathbf{Z}_{k+1}^n}$ and $\pi{\downarrow}_{\mathbf{Y}}$ respectively, then the updated ψ_k evaluates to $\neg\pi[z_k]$ for the same assignment. An easy way to achieve this is to simply add the minterm corresponding to $(\mathbf{Z}_{k+1}^n, \mathbf{Y}) = \pi{\downarrow}_{(\mathbf{Z}_{k+1}^n, \mathbf{Y})}$ to δ_k (resp. γ_k). However, we can do better! Lemma 2 tells us that the value of $\mathbf{Z}_{k+1}^n$, as obtained from the updated candidate Skolem function vector, equals $\pi{\downarrow}_{\mathbf{Z}_{k+1}^n}$ when $\mathbf{Y}$ is set to $\pi{\downarrow}_{\mathbf{Y}}$, regardless of what expansion-based algorithm we use. Therefore, it suffices to simply add the minterm corresponding to $(\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}})$ to δ_k (respectively γ_k) in order to expand it. This motivates Algorithm ExpandAtK shown below. This algorithm takes as input $k = \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$ and expands either δ_k or γ_k, depending on whether ψ_k is chosen to be δ_k or $\neg\gamma_k$. The notation $\delta_k \vee (\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}})$ is used to denote a function that evaluates to 1 when either δ_k evaluates to 1 or when $\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}}$. A similar interpretation applies to $\gamma_k \vee (\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}})$. The expansion of δ_k

or γ_k is accompanied by a corresponding update of ψ_k in lines 3 and 6. Algorithm ExpandAtK also updates the evidence for the counterexample $\pi{\downarrow}_{\mathbf{Y}}$ that results due to the above expansion. Note that $\pi{\downarrow}_{\mathbf{Y}}$ may no longer be a counterexample after the expansion. In this case, the updated value of $\pi{\downarrow}_{\mathbf{Z}}$ simply gives the values of the correct Skolem functions when $\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}}$. If, however, $\pi{\downarrow}_{\mathbf{Y}}$ continues to be a counterexample with a reduced value of the critical index, the updated value of $\pi{\downarrow}_{\mathbf{Z}}$ gives the updated evidence for the counterexample.

Algorithm 3: ExpandAtK

Input: π, k, $(\delta_i, \gamma_i, \psi_i)$ for $1 \le i \le n$
Output: Updated δ_k, γ_k, ψ_k and updated π
`// Requires:` $\pi \models \varepsilon_{\mathbf{\Psi}}$; $k = \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$; ψ_i `is either` δ_i `or` $\neg\gamma_i$ `for` $1 \le i \le n$
1 **if** ψ_k *is* δ_k **then**
2     $\delta_k \leftarrow \delta_k \vee (\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}})$;   `// Expand` δ_k `so it evaluates to` 1 `for` $\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}}$;
3     $\psi_k \leftarrow \delta_k$;
4 **else**
5     $\gamma_k \leftarrow \gamma_k \vee (\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}})$;   `// Expand` γ_k `so it evaluates to` 1 `for` $\mathbf{Y} = \pi{\downarrow}_{\mathbf{Y}}$;
6     $\psi_k \leftarrow \neg\gamma_k$;
`// Now update evidence for` $\pi{\downarrow}_{\mathbf{Y}}$
7 $\pi[z_k] \leftarrow \neg\pi[z_k]$;
8 **for** $j = k-1$ *downto* 1 **do**
9     $\pi[z_j] = \psi_j(\pi{\downarrow}_{\mathbf{Z}^n_{j+1}}, \pi{\downarrow}_{\mathbf{Y}})$;
10 **return** $(\delta_k, \gamma_k, \psi_k)$ and π;

Lemma 5 *The following statements hold after algorithm* ExpandAtK *terminates, where primed versions refer to updated values, assignments and functions at the end of execution of the algorithm.*

1. $\pi'[z_i] = \psi'_i(\pi'{\downarrow}_{\mathbf{Z}^n_{i+1}}, \pi'{\downarrow}_{\mathbf{Y}})$ *for* $1 \le i \le n$.
2. $\kappa_{\mathbf{\Psi}'}(\pi'{\downarrow}_{\mathbf{Y}}) < k$.

Proof To prove part (1), note that Algorithm ExpandAtK updates exactly one candidate Skolem function, i.e. ψ_k. Therefore, by Lemma 2, $\pi'[z_i] = \pi[z_i] = \psi_i(\pi{\downarrow}_{\mathbf{Z}^n_{i+1}}, \pi{\downarrow}_{\mathbf{Y}}) = \psi'_i(\pi'{\downarrow}_{\mathbf{Z}^n_{i+1}}, \pi'{\downarrow}_{\mathbf{Y}})$ for $k < i \le n$. The expansion in lines 1-6 of ExpandAtK, in conjunction with Lemma 3, ensures that the value of $\psi'_k(\pi'{\downarrow}_{\mathbf{Z}^n_{k+1}}, \pi'{\downarrow}_{\mathbf{Y}})$ is the negation of that of $\psi_k(\pi{\downarrow}_{\mathbf{Z}^n_{k+1}}, \pi{\downarrow}_{\mathbf{Y}})$. This, along with the assignment in line 7, ensures that after Algorithm ExpandAtK terminates, $\pi'[z_k] = \psi'_k(\pi'{\downarrow}_{\mathbf{Z}^n_{k+1}}, \pi'{\downarrow}_{\mathbf{Y}})$. The assignment in line 9 ensures that $\pi'[z_i]$ matches $\psi'_i(\pi'{\downarrow}_{\mathbf{Z}^n_{i+1}}, \pi'{\downarrow}_{\mathbf{Y}})$ for $1 \le i < k$.

To prove part (2), note that after the flipping of $\pi[z_k]$ in line 7, we have $\exists \mathbf{Z}_1^{k-1} F(\mathbf{Z}_1^{k-1}, \pi'{\downarrow}_{\mathbf{Z}^n_k}, \pi'{\downarrow}_{\mathbf{Y}}) = 1$, as discussed above. Therefore, $\kappa_{\mathbf{\Psi}'}(\pi'{\downarrow}_{\mathbf{Y}})$ cannot be k or more. If $\pi'{\downarrow}_{\mathbf{Y}}$ ceases to be a counterexample, $\kappa_{\mathbf{\Psi}'}(\pi'{\downarrow}_{\mathbf{Y}}) = 0$. Otherwise, $0 < \kappa_{\mathbf{\Psi}'}(\pi'{\downarrow}_{\mathbf{Y}}) < k$. In either case, the lemma is proved. □

5.2.3 Expansion based on counterexample generalization

The final expansion-based algorithm is inspired by and adapted from the work of John et al. [28]. In their work, the relational specification is assumed to be given in a factored form, i.e. as a conjunction of sub-specifications. They then compute initial under-approximations δ_i and γ_i of Δ_i and Γ_i respectively. Candidate Skolem functions are always chosen to be $\neg\gamma_i$, and satisfying assignments (if any) of the error formula are used to iteratively expand δ_i and γ_i in a CEGAR-like loop. A key component of the algorithm is a sub-routine called UpdateAbsRef [28] that generalizes a counterexample π and uses the generalization to expand δ_i and γ_i for a set of indices i. The termination and correctness proofs of the algorithm in [28] are contingent on the assumption that the specification is given in a factored form, and that candidate Skolem functions ψ_i are always $\neg\gamma_i$. In this paper, we relax these assumptions and show that the basic idea of the algorithm of John et al. [28] can be used in a much more general setting.

Algorithm GeneralizeAndExpand, shown as Algorithm 4, presents our adaptation of Algorithm UpdateAbsRef from [28]. Despite similarities between the two algorithms, there are important differences. For example, the input specification is no longer required to be in factored form and the candidate Skolem function ψ_i is no longer required to be $\neg\gamma_i$. In fact, Algorithm GeneralizeAndExpand requires no additional pre-condition beyond the usual ones.

Algorithm 4: GeneralizeAndExpand

Input: π, k, $(\delta_i, \gamma_i, \psi_i)$ for $1 \leq i \leq n$
Output: Updated $(\delta_i, \gamma_i, \psi_i)$ for $i \in \{1, \ldots \kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})\}$
`// Requires:` $\pi \models \varepsilon_{\mathbf{\Psi}}$; $k = \kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$; `each` ψ_i `is either` δ_i `or` $\neg\gamma_i$

$\ell \leftarrow \max\{m \mid \pi{\downarrow_{(\mathbf{Z}^n_{m+1}, \mathbf{Y})}} \models \delta_m \wedge \gamma_m\}$;
$\mu_0 \leftarrow$ Generalize$(\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1}, \mathbf{Y})}}, \delta_\ell)$;
$\mu_1 \leftarrow$ Generalize$(\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1}, \mathbf{Y})}}, \gamma_\ell)$;
$\mu \leftarrow \mu_0 \wedge \mu_1$;
$\ell \leftarrow \ell + 1$;
`// Loop Invariant:` $\pi{\downarrow_{(\mathbf{Z}^n_{\ell}, \mathbf{Y})}} \models \mu$; $\mathsf{sup}(\mu) \subseteq \mathbf{Z}^n_\ell \cup \mathbf{Y}$; $\mu \Rightarrow \delta_{\ell-1} \wedge \gamma_{\ell-1}$
while $\ell \leq \kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$ **do**
    if $z_\ell \in \mathsf{sup}(\mu)$ **then**
        if $\pi[z_\ell] = 1$ **then**
            $\mu_1 \leftarrow \mu|_{z_\ell = 1}$;
            $\gamma_\ell \leftarrow \gamma_\ell \vee \mu_1$;
            if $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1}, \mathbf{Y})}} \models \delta_\ell$ **then**
                $\mu_0 \leftarrow$ Generalize$(\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1}, \mathbf{Y})}}, \delta_\ell)$;
                $\mu \leftarrow \mu_0 \wedge \mu_1$;
            else
                `break`;
        else
            $\mu_0 \leftarrow \mu|_{z_\ell = 0}$;
            $\delta_\ell \leftarrow \delta_\ell \vee \mu_0$;
            if $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1}, \mathbf{Y})}} \models \gamma_\ell$ **then**
                $\mu_1 \leftarrow$ Generalize$(\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1}, \mathbf{Y})}}, \gamma_\ell)$;
                $\mu \leftarrow \mu_0 \wedge \mu_1$;
            else
                `break`;
    $\ell \leftarrow \ell + 1$;
return $(\delta_i, \gamma_i, \psi_i)$ for $i \in \{1, \ldots \kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})\}$

The basic intuition behind Algorithm GeneralizeAndExpand is as follows. Suppose $\pi \models \varepsilon_{\Psi}$. This yields a single counterexample, viz. $\pi{\downarrow}_{\mathbf{Y}}$, and its corresponding evidence, viz. $\pi{\downarrow}_{\mathbf{Z}}$. We wish to generalize π to a set of assignments, such that each assignment yields a counterexample and the corresponding evidence. Following standard convention, we represent a set of assignments by its characteristic function, i.e. a Boolean function that evaluates to 1 only for assignments in the set. Therefore, we generalize π by a suitably constructed Boolean function μ. In general, it is not necessary for the support of μ to include the whole of $\mathbf{Z} \cup \mathbf{Y}$. Instead, we require that $\mathsf{sup}(\mu) \subseteq \mathbf{Z}_{i+1}^{n} \cup \mathbf{Y}$ for some $i \in \{1, \ldots n\}$, and $\pi{\downarrow}_{(\mathbf{Z}_{i+1}^{n}, \mathbf{Y})} \models \mu$ (hence, μ generalizes π). In order to ensure that every satisfying assignment (not just $\pi{\downarrow}_{(\mathbf{Z}_{i+1}^{n}, \mathbf{Y})}$) of μ yields a counterexample and evidence, we also require that $\mu \Rightarrow (\delta_i \wedge \gamma_i)$. Since $\delta_i \Rightarrow \Delta_i$ and $\gamma_i \Rightarrow \Gamma_i$, this implies that $\mu \models \Delta_i \wedge \Gamma_i$. By definition, $\Delta_i \wedge \Gamma_i \Leftrightarrow \neg\exists \mathbf{Z}_1^i F(\mathbf{Z}_1^i, \mathbf{Z}_{i+1}^n, \mathbf{Y})$. Recalling that $\mathsf{sup}(\mu) \subseteq \mathbf{Z}_{i+1}^{n} \cup \mathbf{Y}$, we conclude that no satisfying assignment of μ can render F true, regardless of what we assign to $\mathbf{Z}_1^i$. Therefore, for every satisfying assignment τ of μ, it is desirable to modify ψ_{i+1} so that it evaluates to $\neg\tau[z_{i+1}]$ whenever $\mathbf{Z}_{i+2}^n$ and $\mathbf{Y}$ are set to $\tau{\downarrow}_{\mathbf{Z}_{i+2}^n}$ and $\tau{\downarrow}_{\mathbf{Y}}$ respectively.

The co-factor $\mu|_{z_{i+1}=1}$ is the characteristic function of the set of assignments of $\mathbf{Z}_{i+2}^n \cup \mathbf{Y}$ that, along with $z_{i+1} = 1$, satisfy μ, thereby preventing F from being satisfied. For all such assignments of $\mathbf{Z}_{i+2}^n \cup \mathbf{Y}$, we need to ensure that ψ_{i+1} (yielding the value of z_{i+1}) doesn't evaluate to 1. This implies that we must expand γ_{i+1} so that it evaluates to 1 whenever $\mu|_{z_{i+1}=1}$ is satisfied. One way of achieving this is to disjoin $\mu|_{z_{i+1}=1}$ with the current γ_{i+1} to obtain the expanded γ_{i+1}. In a similar manner, $\mu|_{z_{i+1}=0}$ can be disjoined with the current δ_{i+1} to obtain an expanded δ_{i+1}. While both δ_{i+1} and γ_{i+1} can indeed be expanded using a generalization of π in this manner, our experiments indicate that this can lead to significant blow-up of memory and time requirements in many cases. Therefore, we choose to expand only one of δ_{i+1} and γ_{i+1}, depending on whether $\pi[z_{i+1}]$ is 0 or 1 respectively. Note that if $\pi[z_{i+1}] = 0$ (resp. $\pi[z_{i+1} = 1)$, we know that $\mu|_{z_{i+1}=0}$ (resp. $\mu|_{z_{i+1}=1}$) indeed has a satisfying assignment, viz. $\pi{\downarrow}_{(\mathbf{Z}_{i+2}^{n}, \mathbf{Y})}$. Therefore, it is reasonable to choose to expand δ_{i+1} (resp. γ_{i+1}) in this case.

The above strategy of expanding δ_{i+1} and/or γ_{i+1} results in updation of the candidate Skolem function ψ_{i+1}. However, even after this expansion, it may turn out that $\pi{\downarrow}_{(\mathbf{Z}_{i+2}^{n}, \mathbf{Y})}$ satisfies $\delta_{i+1} \wedge \gamma_{i+1}$. If this happens, we can repeat the above argument with $i+1$ substituted for i. This suggests an iterative procedure for expanding δ_j and/or γ_j for increasing values of j in $\{i+1, \ldots n\}$. The iteration is stopped when $\pi{\downarrow}_{(\mathbf{Z}_{j+1}^{n}, \mathbf{Y})}$ no longer satisfies $\delta_j \wedge \gamma_j$. Since $k = \kappa_{\Psi}(\pi{\downarrow}_{\mathbf{Y}})$, we already know that $\exists \mathbf{Z}_1^k F(\mathbf{Z}_1^k, \pi{\downarrow}_{\mathbf{Z}_{k+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$. Therefore, $\pi{\downarrow}_{(\mathbf{Z}_{k+1}^{n}, \mathbf{Y})} \not\models \delta_k \wedge \gamma_k$, and the above iterative procedure can be terminated early at k, instead of iterating up to n.

The pseudocode in Algorithm 4 formalizes the intuition described above. The algorithm starts off by identifying the largest index $\ell \in \{1, \ldots n\}$ such that $\pi \models \delta_\ell \wedge \gamma_\ell$. It then generalizes π to a formula μ with support in $\mathbf{Z}_{\ell+1}^n \cup \mathbf{Y}$ such that $\pi{\downarrow}_{(\mathbf{Z}_{\ell+1}^{n}, \mathbf{Y})} \models \mu$ and $\mu \Rightarrow \delta_\ell \wedge \gamma_\ell$. This is done using a sub-routine Generalize (discussed later) in lines 2-4 of Algorithm GeneralizeAndExpand. After ℓ is incremented in line 5, the loop in lines 16-24 maintains the following three invariants at the loop head: (a) $\pi{\downarrow}_{(\mathbf{Z}_{\ell}^{n}, \mathbf{Y})} \models \mu$, (b) $\mathsf{sup}(\mu) \subseteq \mathbf{Z}_\ell^n \cup \mathbf{Y}$, and (c) $\mu \Rightarrow \delta_{\ell-1} \wedge \gamma_{\ell-1}$. There are two ways in which the loop eventually terminates: (a) either ℓ, which is incremented in every iteration (line 24), exceeds $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$, or (b) we detect that $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}}$ no longer satisfies $\delta_\ell \wedge \gamma_\ell$ in the body of the loop (lines 15 and 23).

Within the body of the loop, if the condition in line 8 holds, we know that $\pi[z_\ell] = 1$. Additionally, from the loop invariant, we know that $\pi{\downarrow_{(\mathbf{Z}^n_\ell,\mathbf{Y})}} \models \mu$. It follows that $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}} \models \mu_1$ at line 10, where $\mu_1 = \mu|_{z_\ell=1}$. Therefore, μ_1 serves as a generalization of $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}}$. Note also that $\mu \Rightarrow \delta_{\ell-1} \wedge \gamma_{\ell-1}$ (loop invariant) and $\delta_{\ell-1} \wedge \gamma_{\ell-1} \;\Rightarrow\; \Delta_{\ell-1} \wedge \Gamma_{\ell-1}$ by definition. Therefore, $\mu_1 \Rightarrow (\Delta_{\ell-1} \wedge \Gamma_{\ell-1})|_{z_\ell=1}$. However, $(\Delta_{\ell-1} \wedge \Gamma_{\ell-1})|_{z_\ell=1} \Leftrightarrow \Gamma_\ell$ by the definitions of $\Delta_{\ell-1}$, $\Gamma_{\ell-1}$ and Γ_ℓ. Hence, $\mu_1 \Rightarrow \Gamma_\ell$ and we can safely expand γ_ℓ by disjoining it with μ_1. This is exactly what Algorithm GeneralizeAndExpand does in line 10. Clearly, $\mu_1 \Rightarrow \gamma_\ell$ after the statement in line 10 is executed.

In line 11, we check if $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}} \models \delta_\ell$ holds. If so, we have $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}} \models \gamma_\ell \wedge \delta_\ell$, since we already knew that $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}} \models \gamma_\ell$ after the statement in line 10 was executed. In this case, we use the Generalize sub-routine to obtain a formula μ_0 with support in $\mathbf{Z}^n_{\ell+1} \cup \mathbf{Y}$ such that $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}} \models \mu_0$ and $\mu_0 \Rightarrow \delta_\ell$. It is now straightforward to see that the formula $\mu_0 \wedge \mu_1$, with support in $\mathbf{Z}^n_{\ell+1} \cup \mathbf{Y}$, generalizes $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}}$, while under-approximating $\delta_\ell \wedge \gamma_\ell$. Thus, the loop invariant is satisfied with ℓ being replaced by $\ell+1$, and we can proceed to the next iteration of the loop. If, on the other hand, the check in line 11 fails, then $\pi{\downarrow_{(\mathbf{Z}^n_{\ell+1},\mathbf{Y})}} \not\models \gamma_\ell \wedge \delta_\ell$, and the loop invariant would be violated if we continued with the next iteration after incrementing ℓ. Therefore, we exit the loop in line 15.

The above discussion considered the case when $\pi[z_\ell] = 1$. If $\pi[z_\ell] = 0$, the check in line 8 fails and the statements in lines 17-23 are executed. These statements achieve a similar effect as discussed above, except that δ_ℓ is updated instead of γ_ℓ. Of course, the above discussion is meaningful only if $z_\ell \in \mathsf{sup}(\mu)$. The check in line 7 ensures that this condition holds before we proceed to update δ_ℓ and/or γ_ℓ.

For the function Generalize, there are several options for implementing it. In general, given $\pi{\downarrow_{(\mathbf{Z}^n_{j+1},\mathbf{Y})}} \models g$, where $\mathsf{sup}(g) \subseteq \mathbf{Z}^n_{j+1} \cup \mathbf{Y}$, we require Generalize$(\pi{\downarrow_{(\mathbf{Z}^n_{j+1},\mathbf{Y})}}, g)$ to return a Boolean function g' with support in $\mathbf{Z}^n_{j+1} \cup \mathbf{Y}$ such that $\pi{\downarrow_{(\mathbf{Z}^n_{j+1},\mathbf{Y})}} \models g'$ and $g' \Rightarrow g$ holds. At one extreme, we can return the minterm corresponding to $\pi{\downarrow_{(\mathbf{Z}^n_{j+1},\mathbf{Y})}}$ as g'. While this gives a correct implementation of Generalize, it doesn't really generalize the counterexample, and the benefits of generalization are lost. At the other extreme, we can return g itself as the result. While this achieves the purpose of generalizing a counterexample, our experiments indicated that the memory and time overheads of this option are too high in our context. So we adopt a middle path as follows. As in [28], we use a set of implicitly disjoined formulas to represent each δ_i and γ_i. If g is δ_j or γ_j, we let Generalize$(\pi{\downarrow_{(\mathbf{Z}^n_{j+1},\mathbf{Y})}}, g)$ return one of the formulas, say g_i, in the above set – specifically, the one with the smallest support such that $\pi{\downarrow_{(\mathbf{Z}^n_{j+1},\mathbf{Y})}} \models g_i$. For reasons of practical performance, we restrict the sizes of individual formulas in the set of implicitly disjoined formulas to be in $\mathcal{O}(|F|)$. Note that this can always be done since the minterm corresponding to $\pi{\downarrow_{(\mathbf{Z}^n_{j+1},\mathbf{Y})}}$ is of size $|Y|$, and hence is in $\mathcal{O}(|F|)$.

Lemma 6 *The following statements hold for Algorithm* GeneralizeAndExpand.

1. *The index ℓ computed in line* 1 *lies in* $\{1, \ldots \kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}}) - 1\}$.
2. *There are three loop invariants at line* 6*: (i)* $\pi{\downarrow_{(\mathbf{Z}^n_\ell,\mathbf{Y})}} \models \mu$*, (ii)* $\mathsf{sup}(\mu) \subseteq \mathbf{Z}^n_\ell \cup \mathbf{Y}$ *and (iii)* $\mu \Rightarrow \delta_{\ell-1} \wedge \gamma_{\ell-1}$.

3. There is at least one $\ell \in \{2, \ldots \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})\}$ such that either δ_ℓ or γ_ℓ is expanded.

Proof To prove part (1), we first show that $\pi{\downarrow}_{(\mathbf{Z}_2^n,\mathbf{Y})} \models \delta_1 \wedge \gamma_1$. Since $\pi \models \varepsilon_{\mathbf{\Psi}}$, we know that $F(\pi{\downarrow}_{\mathbf{Z}}, \pi{\downarrow}_{\mathbf{Y}}) = 0$ and $\pi[z_1] = \psi_1(\pi{\downarrow}_{\mathbf{Z}_2^n}, \pi{\downarrow}_{\mathbf{Y}})$. Now recall from Section 5.2.1 that $\delta_1 \Leftrightarrow \Delta_1$ and $\gamma_1 \Leftrightarrow \Gamma_1$. Therefore, regardless of whether $\psi_1 = \delta_1$ or $\psi_1 = \neg\gamma_1$, the Skolem function ψ_1 is correct for z_1. In other words, if $\exists z_1 F(z_1, \pi{\downarrow}_{\mathbf{Z}_2^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$, then $F\left(\psi_1(\pi{\downarrow}_{\mathbf{Z}_2^n}, \pi{\downarrow}_{\mathbf{Y}}), \pi{\downarrow}_{\mathbf{Z}_2^n}, \pi{\downarrow}_{\mathbf{Y}}\right) = F(\pi{\downarrow}_{\mathbf{Z}}, \pi{\downarrow}_{\mathbf{Y}}) = 1$ as well. However, this contradicts our earlier observation that $F(\pi{\downarrow}_{\mathbf{Z}}, \pi{\downarrow}_{\mathbf{Y}}) = 0$. Therefore, we must have $\exists z_1 F(z_1, \pi{\downarrow}_{\mathbf{Z}_2^n}, \pi{\downarrow}_{\mathbf{Y}}) = 0$. From the definitions of δ_1 and γ_1, this implies that $\pi{\downarrow}_{(\mathbf{Z}_2^n,\mathbf{Y})} \models \delta_1 \wedge \gamma_1$. Therefore, if $\ell = \max\{m \mid \pi{\downarrow}_{(\mathbf{Z}_{m+1}^n,\mathbf{Y})} \models \delta_m \wedge \gamma_m\}$, then $\ell \geq 1$.

Let $k = \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$. Suppose, if possible, $\pi{\downarrow}_{(\mathbf{Z}_{i+1}^n,\mathbf{Y})} \models \delta_i \wedge \gamma_i$ for some $i \in \{k, \ldots n\}$. Since $\delta_i \Rightarrow \Delta_i$ and $\gamma_i \Rightarrow \Gamma_i$, we have $\pi{\downarrow}_{(\mathbf{Z}_{i+1}^n,\mathbf{Y})} \models \Delta_i \wedge \Gamma_i$. By definition of Δ_i and Γ_i, this means $\exists \mathbf{Z}_1^i F(\mathbf{Z}_1^i, \pi{\downarrow}_{\mathbf{Z}_{i+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 0$. However, from Definition 3, we know that for all $i \in \{k \ldots n\}$, $\exists \mathbf{Z}_1^i F(\mathbf{Z}_1^i, \pi{\downarrow}_{\mathbf{Z}_{i+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 1$. This gives a contradiction. Hence, $\pi{\downarrow}_{(\mathbf{Z}_{i+1}^n,\mathbf{Y})} \not\models \delta_i \wedge \gamma_i$ for all $i \in \{k, \ldots n\}$. Therefore, if $\ell = \max\{m \mid \pi{\downarrow}_{(\mathbf{Z}_{m+1}^n,\mathbf{Y})} \models \delta_m \wedge \gamma_m\}$, then $\ell < k = \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$. Combining the lower and upper bounds of ℓ obtained above, we get $\ell \in \{1, \ldots \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}}) - 1\}$.

In order to prove part (2), we need to show that the invariants hold in the following three cases, where line numbers refer to those in the pseudocode for Algorithm GeneralizeAndExpand: (a) after ℓ is incremented in line 5, (b) after ℓ is incremented in line 24 following the updation of μ in line 13, and (c) after ℓ is incremented in line 24 following the updation of μ in line 21. All of these cases have been discussed in detail while describing the steps in Algorithm GeneralizeAndExpand, where it has been argued why the three invariants hold in each of these cases.

To prove part (3), let ℓ_0 be the value of ℓ identified in line 1, and let $k = \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$. As proved in part (1), $1 \leq \ell_0 \leq k - 1$. Therefore, when control reaches line 6 after incrementing ℓ in line 5, we have $2 \leq \ell \leq k$ and the loop in lines 6-24 is executed at least once. We now ask if it is possible for $z_\ell \not\in \mathsf{sup}(\mu)$ for all $\ell \in \{\ell_0 + 1, \ldots k\}$, where μ is as computed in line 4. Suppose, if possible, this is true. Then, by virtue of of the way in which μ_0, μ_1 and μ are calculated in lines 2, 3 and 4, we have $\mathsf{sup}(\mu) \subseteq \mathbf{Z}_{k+1}^n \cup \mathbf{Y}$. We know from the loop invariant at line 6 that $\mu \Rightarrow (\delta_{\ell_0} \wedge \gamma_{\ell_0}) \Rightarrow \Delta_{\ell_0} \wedge \Gamma_{\ell_0}$. Using the definitions of Δ_{ℓ_0} and Γ_{ℓ_0}, we get $\mu \Rightarrow \neg\exists \mathbf{Z}_1^{\ell_0} F(\mathbf{Z}_1^{\ell_0}, \mathbf{Z}_{\ell_0+1}^n, \mathbf{Y})$. Since $\mathsf{sup}(\mu) \subseteq \mathbf{Z}_{k+1}^n \cup \mathbf{Y}$ by assumption and since $\ell_0 + 1 \leq k < k + 1$, the above implication simplifies to $\mu \Rightarrow \neg\exists \mathbf{Z}_1^k F(\mathbf{Z}_1^k, \mathbf{Z}_{k+1}^n, \mathbf{Y})$. Additionally, $\pi{\downarrow}_{(\mathbf{Z}_{\ell_0+1}^n,\mathbf{Y})} \models \mu$ from the loop invariant at line 6. Once again, since $\mathsf{sup}(\mu) \subseteq \mathbf{Z}_{k+1}^n \cup \mathbf{Y}$, this simplifies to $\pi{\downarrow}_{(\mathbf{Z}_{k+1}^n,\mathbf{Y})} \models \mu$. Therefore, we get $\pi{\downarrow}_{(\mathbf{Z}_{k+1}^n,\mathbf{Y})} \models \neg\exists \mathbf{Z}_1^k F(\mathbf{Z}_1^k, \mathbf{Z}_{k+1}^n, \mathbf{Y})$. In other words $\exists \mathbf{Z}_1^k F(\mathbf{Z}_1^k, \pi{\downarrow}_{\mathbf{Z}_{k+1}^n}, \pi{\downarrow}_{\mathbf{Y}}) = 0$. This contradicts the definition of $\kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})$ $(= k)$.

The above argument shows that when control reaches the loophead at line 6 for the first time, there is at least one $\ell \in \{2, \ldots k\}$ such that $z_\ell \in \mathsf{sup}(\mu)$. Hence, either line 10 or line 18 is executed, resulting in updation of either δ_ℓ or γ_ℓ, for some $\ell \in \{2, \ldots \kappa_{\mathbf{\Psi}}(\pi{\downarrow}_{\mathbf{Y}})\}$. This proves part (3) of the lemma. $\square$

Algorithm 5: PHASE2

Input: F, c, $(\delta_i, \gamma_i, \psi_i)$ for all $i \in \{1, \ldots n\}$
Output: Correct (updated) Skolem functions ψ_i for all $i \in \{1, \ldots n\}$
`// Requires: For all` $i \in \{1, \ldots n\}$, δ_i, γ_i `are as obtained from Phase 1`
`// Requires: For all` $i \in \{1, \ldots n\}$, ψ_i `is either` δ_i `or` $\neg\gamma_i$
1 Initialize Skolem functions as in Eqn (4);
2 **while** $\varepsilon_{\mathbf{\Psi}}$ *is satisfiable* **do**
3   Let π be an assignment s.t. $\pi \models \varepsilon_{\mathbf{\Psi}}$;   `// Use a SAT solver;`
4     $k \leftarrow$ COMPUTEK$(\pi, \mathbf{\Psi})$;
5   **while** $k \neq 0$ **do**
        `//` $\pi{\downarrow_{\mathbf{Y}}}$ `is still a counterexample for` $\mathbf{\Psi}$`;`
6   **if** $0 \leq k \leq c$ **then**
7   MAXEXPAND(c, δ_1, γ_1);
8   break;   `// Guaranteed to happen at most once;`
9   GENERALIZEANDEXPAND$(\pi, k, \{\delta_i, \gamma_i, \psi_i \mid 1 \leq i \leq n\})$;
10   EXPANDATK$(\pi, k, \{\delta_i, \gamma_i, \psi_i \mid 1 \leq i \leq n\})$; `// Also updates evidence in` π`;`
11         $k \leftarrow$ COMPUTEK$(\pi, \mathbf{\Psi})$;
12 **return** ψ_i for all $i \in \{1, \ldots n\}$;

5.2.4 Combining three expansion-based algorithms

While each of the three expansion-based algorithms presented above can be used, either repeatedly and/or with specific choices of parameters, to eliminate all counterexamples and obtain a correct Skolem function vector, our experiments indicate that a hybrid of the three algorithms outperforms any one of them individually. This hybrid algorithm, shown as Algorithm 5, constitutes phase 2 of BFSS.

The algorithm is parametrized with $c \in \{1, \ldots n\}$, which is used for MAXEXPAND. In practice, we use a small value of c, viz. 4, and maximally expand δ_i and γ_i for all $i \in \{1, \ldots c\}$ if $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$ is small and happens to lie in $\{1, \ldots c\}$. Note that an invocation of MAXEXPAND is guaranteed to happen at most once. This is because once it is invoked, the Skolem functions $\psi_1, \ldots \psi_c$ are guaranteed to be correct, and hence $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$ necessarily exceeds c if $\pi{\downarrow_{\mathbf{Y}}}$ is a counterexample. We use GENERALIZEANDEXPAND to first expand a set of δ_i and/or γ_i using a generalization of $\pi{\downarrow_{\mathbf{Y}}}$. Then we use EXPANDATK to ensure that the critical index of the candidate Skolem function vector w.r.t. the current counterexample strictly reduces regardless of the expansion(s) effected by GENERALIZEANDEXPAND. Recall that an invocation of EXPANDATK also updates π, especially the evidence $\pi{\downarrow_{\mathbf{Z}}}$. Sub-routine COMPUTEK is then invoked to determine the critical index of the updated $\mathbf{\Psi}$ with respect to the current counterexample $\pi{\downarrow_{\mathbf{Y}}}$. Once $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$ becomes 0, we know that $\pi{\downarrow_{\mathbf{Y}}}$ is no longer a counterexample, and can never surface again as a counterexample. The error formula $\varepsilon_{\mathbf{\Psi}}$ is then re-computed with the updated candidate Skolem function vector $\mathbf{\Psi}$, and the next iteration of the outer loop in lines 2-11 started. If $\varepsilon_{\mathbf{\Psi}}$ is unsatisfiable, we know by Theorem 3 that we have a correct Skolem function vector. Otherwise, a satisfying assignment π of $\varepsilon_{\mathbf{\Psi}}$ is obtained (line 3), the critical index updated (line 4) and the inner loop in lines 5-11 executed again.

Theorem 5 *The following statements hold for Algorithm* PHASE2.

1. *It terminates when invoked with δ_i, γ_i and ψ_i as generated by phase 1 of* BFSS.

2. *On termination, it produces a correct Skolem function vector.*
3. *The worst-case size of a Skolem function ψ_i is in $\mathcal{O}\big(|F| \cdot 2^{|\mathbf{Y}|}\big)$.*

Proof To prove part (1), note that the only sub-routines in Algorithm Phase2 that modify δ_i and/or γ_i and, hence ψ_i, are MaxExpand, ExpandAtK and GeneralizeAndExpand. All of these are expansion-based algorithms. Therefore, by Corollary 2, once a counterexample is eliminated by Algorithm Phase2, it cannot be re-introduced.

Every iteration of the inner loop in lines 5-11 of Algorithm 5 either results in an invocation of ExpandAtK or an invocation of MaxExpand. Since MaxExpand is invoked only when $1 \leq \kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}}) \leq c$, it follows from Lemma 4 that the counterexample $\pi{\downarrow_{\mathbf{Y}}}$ is eliminated by the invocation in one go. If, on the other hand, ExpandAtK is invoked, then by virtue of Lemma 5(2), there is a strict reduction of $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$. Hence, after at most n iterations of the inner loop, $\kappa_{\mathbf{\Psi}}(\pi{\downarrow_{\mathbf{Y}}})$ must become 0. By Proposition 4, the counterexample $\pi{\downarrow_{\mathbf{Y}}}$ is eliminated in at most n iterations of the inner loop. The total number of iterations of the outer loop (lines 2-11) of Algorithm 5 is at most M, where M is the count of counterexamples (i.e. $\pi{\downarrow_{\mathbf{Y}}}$) for the candidate Skolem function vector obtained from phase 1 of bfss. Overall, Algorithm 5 terminates after $\mathcal{O}(M \cdot n)$ steps, where each step may involve $\mathcal{O}(\log_2 n)$ invocations of an NP-oracle in sub-routine ComputeK.

To prove part (2), note that the outer loop in lines 3-11 terminates only when $\varepsilon_{\mathbf{\Psi}}$ becomes unsatisfiable. By virtue of Theorem 3, the Skolem function vector returned by Algorithm Phase2 on termination is indeed correct.

To prove part (3), note that M alluded to above refers to the count of counterexamples for the candidate Skolem function vector obtained from phase 1 of bfss. Since this can be as large as $2^{|\mathbf{Y}|}$, the number of times each δ_i and/or γ_i can undergo expansion is at most $2^{|\mathbf{Y}|}$.

If the expansion happens in MaxExpand, it can result in a blow-up of candidate Skolem function sizes by a factor of 2^c. In general, c can be as large as $|\mathbf{Y}|$. However, since MaxExpand can be invoked at most once in any run of Algorithm Phase2, it contributes at most a $2^{\mathcal{O}(|\mathbf{Y}|)}$ factor blow-up in sizes of candidate Skolem functions. In practice, we cap c at a small value, viz. 4. Hence, the blow-up in sizes of candidate Skolem functions due to expansion in MaxExpand is limited to a constant factor in practice. Every expansion in ExpandAtK effectively adds a minterm corresponding to the counterexample $\pi{\downarrow_{\mathbf{Y}}}$ to either δ_i or γ_i. Hence, the increase in size of a candidate Skolem function due to an expansion effected by ExpandAtK is in $\mathcal{O}(|\mathbf{Y}|)$. If the expansion happens in GeneralizeAndExpand, note from the pseudocode in Algorithm 4 that either $\mu|_{z_\ell=0}$ or $\mu|_{z_\ell=1}$ is added to δ_ℓ or γ_ℓ respectively (see lines 10 and 18 of Algorithm 4). Recall also that μ is obtained as the conjunction of μ_0 and μ_1, where μ_0 and μ_1 are computed by function Generalize. Our choice of Generalize, discussed earlier, ensures that the sizes of μ_0 and μ_1 are in $\mathcal{O}(|F|)$. Therefore, the potential increase in size of a candidate Skolem function due to a single invocation of GeneralizeAndExpand is in $\mathcal{O}(|F|)$.

Since $\mathcal{O}(|\mathbf{Y}|)$ is subsumed by $\mathcal{O}(|F|)$, it follows from the above discussion that that the size of δ_i and/or γ_i, and hence of ψ_i, when Algorithm Phase2 terminates is in $\mathcal{O}\big(|F| \cdot 2^{|\mathbf{Y}|}\big)$. $\square$

As part of additional explorations, we also experimented with a variant of Algorithm Phase2 that sampled multiple counterexamples from the set of satisfying assignments of $\varepsilon_{\mathbf{\Psi}}$ using a state-of-the-art almost uniform sampler [13]. The intent of using this variant was to allow Phase2 to benefit from choosing a "good" counterexample from a set of counterexamples, instead of using the only one returned by a SAT solver in line 3. In this variant, we invoked MaxExpand with parameter c if any of the sampled counterexamples had $\kappa_{\mathbf{\Psi}}(\pi\downarrow_{\mathbf{Y}}) \leq c$, and invoked GeneralizeAndExpand and RefineAtK with the counterexample that yielded the maximum ℓ, as computed in line 1 of GeneralizeAndExpand. Extensive experiments however failed to indicate any performance gains compared to Algorithm 5. Therefore, we omit discussing this variant in this paper.

6 Experimental results

We have implemented Algorithm BFSS and done extensive experimentation to compare its performance with that of several state-of-the-art Boolean functional synthesis tools. In Subsection 6.1, we describe our experimental setup, the benchmark suites and the implementation architecure. Next, in Subsection 6.2, we present our experimental results and analyze the performance of bfss. Finally, in Section 6.3, we compare the performance of bfss with several state-of-the-art tools.

6.1 Methodology

Our implementation consists of two parallel pipelines that accept the same input specification but represent them in two different ways. The first pipeline takes the input formula as an AIG and builds an NNF DAG (not necessarily in wDNNF) – we call this the AIG-NNF pipeline. The second pipeline builds an ROBDD from the input AIG using dynamic variable reordering (no restrictions on variable order), and then obtains a DNNF (and hence wDNNF) representation from it using the linear-time algorithm described in [17]. We call this the BDD pipeline. Once the DAG representation of F is built, we use Algorithm 1 on both the representations to generate Skolem functions. In the case of the AIG-NNF pipeline, if phase 1 does not give the correct Skolem functions, we use phase 2. In the case of the BDD pipeline, however, we know from Theorem 4(2) that there is no need to invoke phase 2. For discussions in this section, we call the *ensemble of AIG-NNF and BDD pipelines* bfss. Note that they only differ in the representation of the specification $F(\mathbf{X}, \mathbf{Y})$.

Our implementation of bfss uses the ABC [10] library with MiniSAT to represent and manipulate Boolean functions. We compare bfss with the following tools for Boolean functional synthesis: (*i*) parSyn [1], (*ii*) Cadet [38], (*iii*) BaFSyn [14] and (*iv*) AbsSynSkolem (based on the BFnS step of AbsSynthe [11]).

We consider a total of 523 benchmarks, taken from four different domains:

(a) 48 *Arithmetic benchmarks* from [19], with varying bit-widths (viz. 32, 64, 128, 256, 512 and 1024) of arithmetic operators,
(b) 68 *Disjunctive Decomposition benchmarks* from [1], generated by considering some of the larger HWMCC10 benchmarks,
(c) 5 *Factorization benchmarks*, also from [1], representing factorization of numbers of different bit-widths (8, 10, 12, 14, 16), and

Benchmark Domain	Total Benchmarks	Solved by AIG-NNF Pipeline	Solved By BDD Pipeline	Total Solved by BFSS
QBFEval	402	181	159	**201**
Arithmetic	48	36	36	**45**
Disjunctive Decomposition	68	68	59	**68**
Factorization	5	4	5	**5**
Total	523	289	256	**319**

Table 1: BFSS: Performance at a glance

(d) 402 *QBFEval benchmarks*, taken from the Prenex 2QBF track of QBFEval 2018 [36].

Since different tools accept benchmarks in different formats, each benchmark was converted to both `qdimacs` and `Verilog/Aiger` formats. All benchmarks and the procedure by which we generated (and converted) them are detailed in [3]. We use "balance; rewrite -l; refactor -l; balance; rewrite -l; rewrite -lz; balance; refactor -lz; rewrite -lz; balance" as the ABC script for optimizing the AIG representation of the input specification.

For each benchmark, the order $\preceq$ (ref. step 11 of Algorithm 1) in which Skolem functions are generated is such that if z_i occurs in the transitive fan-in of fewer nodes in the AIG representation of $F(\mathbf{Z}, \mathbf{Y})$ than z_j, then $z_i \preceq z_j$. This order is used for both BFSS and PARSYN. Note that this is unrelated to the dynamic variable order used to construct an ROBDD of the input specification in the BDD pipeline.

All experiments were performed on a message-passing cluster, with 20 cores and 64 GB memory per node, each core being a 2.2 GHz Intel Xeon processor. The operating system was Cent OS 6.5. Twenty cores were assigned to each run of PARSYN, which benefits from using parallel execution. For each of BAFSYN, CADET, ABSSYNSKOLEM and for each of the two pipelines of BFSS, a single core was used, since these tools don't exploit parallelism. The maximum time given for execution of any run was 3600 seconds. The total amount of main memory for any run was restricted to 16GB. The metric used to compare the algorithms was *time taken to synthesize Boolean functions* and the *size* of the synthesized functions. The time reported for BFSS is the better of the two times obtained from the two pipelines described above, which only differ in the representation of the input.

6.2 BFSS performance and a comparison of its two pipelines

We present the results of BFSS in Table 1. Aggregating over the two pipelines mentioned above, BFSS solved 319 benchmarks out of 523. Table 1 also gives the relative performance of the two pipelines at a glance. We now discuss the performance of each of the pipelines in detail.

The AIG-NNF pipeline Table 2 gives the performance summary of the AIG-NNF pipeline. Of the 402 benchmarks in QBFEVAL, the AIG-NNF pipeline solved 181 benchmarks, of which 118 were solved in phase 1. On 14 benchmarks, phase 1 did not terminate in the specified resource contraints. Hence, phase 2 was commenced on the remaining 270 benchmarks, of which 63 benchmarks were solved

Benchmark Domain	Total Benchmarks	# Benchmarks Solved	Solved by Phase 1	Phase 2 Started	Solved By Phase 2	Avg % Of Unate Output Vars
QBFEval	402	181	118	270	63	38.2
Arithmetic	48	36	36	12	0	0
Disjunctive Decomposition	68	68	68	0	0	64.13
Factorization	5	4	0	5	4	0

Table 2: BFSS: Performance Summary for AIG-NNF pipeline

within the specified resource constraints. Of the 118 solved in phase 1, 63 were found to have all output variables unate. Of these, 11 benchmarks had only syntactically detectable unate outputs (i.e. unateness detected by identifying pure literals) and 12 had only semantically detectable unate outputs (i.e. required a satisfiability check of η_i^+ and/or η_i^-, given by Equations (1) and (2)). For the DisjDecomposition benchmark suite, 20 benchmarks were found to contain only unate outputs, of which 19 benchmarks contained semantically detectable unate outputs. The Arithmetic and the Factorization benchmark suite did not have any instance with unate output variables.

We found that benchmarks that contained only unate (syntactically and/or semantically detected) output variables were restricted to certain families in the QBFEval and DisjDecomposition suites. For the QBFEval suite, these included **AR-fixpoint**, **cache-coherence**, **ethernet-fixpoint**, **itc-b13-fixpoint**, **pi-bus-fixpoint**, **small-seq-fixpoint**, **small-synabs-fixpoint** and some of the **stmt** and **usb-phy-fixpoint** families. Similarly, in DisjDecomposition, the **bobsmhdlc**, **bobsynthneg** and **neclaftp** family of benchmarks contained only unate output variables.

We observed that the number of unate output variables detected semantically was higher than those detected syntactically, justifying the need for the semantic unate checks. On average, 38.2% of the output variables in the QBFEval benchmark suite were found to be unate. Of these, on average 15.5% were detected syntactically and 22.7% were detected semantically. Similarly, for DisjDecomposition, 64.13% of the output variables were unate, of which 0.61% were detected syntactically and 63.51% were detected semantically.

Finally, we examined the count of counterexamples required by the AIG-NNF pipeline for the 63 benchmarks in QBFEval that were solved by phase 2 of BFSS. For most of these benchmarks, this count was less than 5. However, about 6 benchmarks required expansion based on > 30 counterexamples, the maximum count being 138.

The BDD pipeline The BDD pipeline solved a total of 256 benchmarks across all four domains (see Table 1). Note that if this pipeline solves a benchmark, it does so by constructing a wDNNF representation from the BDD representation. Hence all the 256 benchmarks solved by the BDD pipeline are in wDNNF by construction. In constrast, we found only 83 of the solved benchmarks in the AIG-NNF pipeline to be in wDNNF. However, note that 222 benchmarks were solved in phase 1 using the AIG-NNF pipeline. This is attributable to specifications satisfying the condition of Theorem 2(a) (while not being in wDNNF). A more detailed study of the representation related issues and analysis has been done recently in [2].

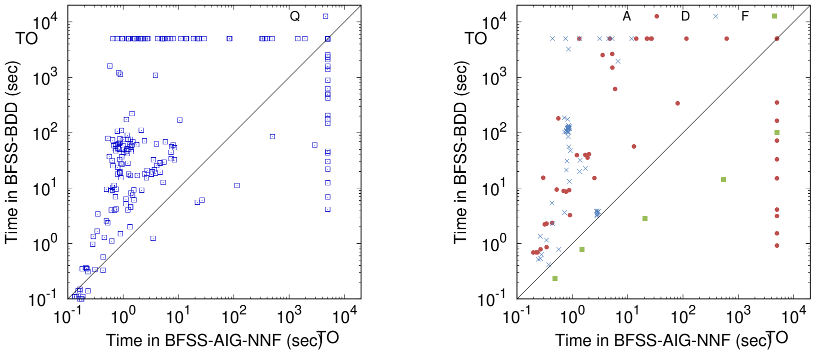

Fig. 1: BFSS: AIG-NNF vs BDD: Time Taken to synthesize Skolem Functions. Legend: Q: QBFEval, A: Arithmetic, F: Factorization, D: DisjDecomposition. TO: benchmarks for which the corresponding algorithm was unsuccessful.

Comparison of the pipelines We now compare the time taken and the size of the Skolem functions generated by the two pipelines. For clarity, since the number of benchmarks in the QBFEval suite is considerably greater, we plot the QBFEval benchmarks separately. Figure 1 shows the results for the time taken by the two pipelines on the QBFEval, Arithmetic, DisjDecomposition and Factorization suite of benchmarks. As can be seen from Figure 1, there are some benchmarks which are solved by only one of the pipelines. But for most of the QBFEval, Arithmetic and DisjDecomposition benchmarks which are solved by both pipelines, the AIG-NNF pipeline takes less time than the BDD pipeline. For the Factorization benchmark suite, the BDD pipeline takes less time.

We next compare the sizes of the Skolem functions generated by the two pipelines. Figure 2 shows a comparison of the average sizes of Skolem functions for QBFEval and Arithmetic, DisjDecomposition and Factorization benchmarks. For every benchmark, the average is calculated over all components of the entire Skolem function vector generated by the algorithm. We observe that for most of the benchmarks that are solved by both the pipelines, the sizes of Skolem Functions generated by the AIG-NNF pipeline are comparable or smaller.

In other words, the AIG-NNF pipeline, in most instances, not only takes less time than the BDD pipeline, it also generates smaller Skolem functions. However, there are instances that are solved exclusively by either the AIG-NNF pipeline or the BDD pipeline; hence we retain both pipelines in our tool.

6.3 Comparison of bfss with other tools

In this section, we compare the performance of bfss with other state-of-the-art tools. Table 3 gives the comparative performance at a glance, in terms of the number of benchmarks solved by the various tools.

bfss *vs* Cadet : Of the 523 benchmarks, Cadet was successful on 254 benchmarks, of which 9 belonged to DisjDecomposition, 28 to Arithmetic, 4 to Factorization

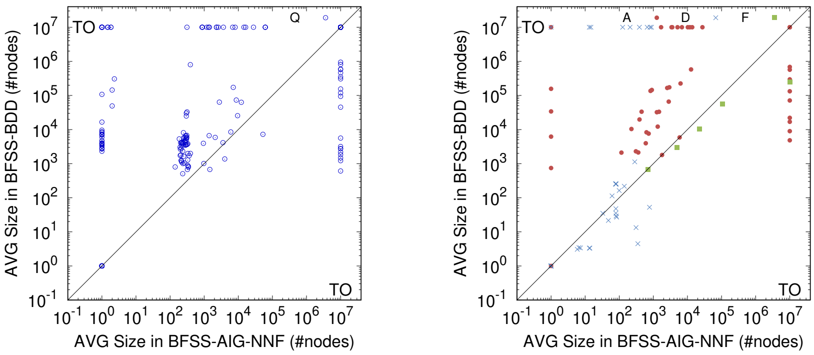

Fig. 2: BFSS: AIG-NNF vs BDD: Avg Sizes of Skolem Functions. Legend: Q: QBFEval, A: Arithmetic, F: Factorization, D: DisjDecomposition. TO: benchmarks for which the corresponding algorithm was unsuccessful.

Benchmark Domain	Total Benchmarks	Solved by BFSS	Solved By Cadet	Solved by parSyn	Solved by AbsSynSkolem	Solved by BaFSyn
QBFEval	402	201	213	118	151	11
Arithmetic	48	45	28	15	32	0
Disjunctive Decomposition	68	68	9	64	29	0
Factorization	5	5	4	3	5	0
Total	**523**	**319**	**254**	**200**	**217**	**11**

Table 3: Number of benchmarks solved by each tool

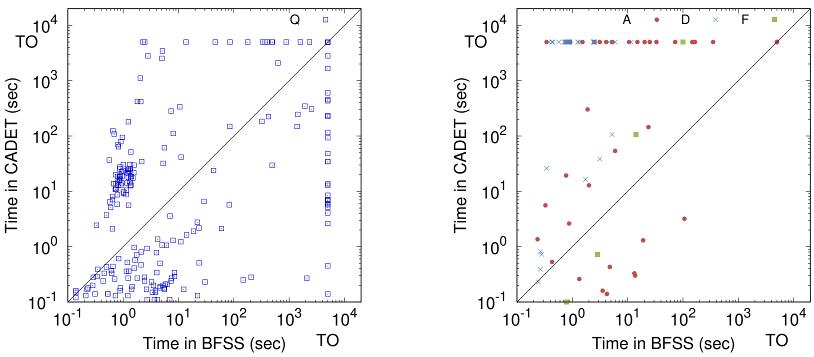

Fig. 3: BFSS vs Cadet: Time Taken to synthesize Skolem Functions. Legend: Please see Figure 1.

and 213 to QBFEval. Figure 3(a) gives the performance of the two algorithms with respect to time on the QBFEval suite. Here, Cadet solved 26 benchmarks that BFSS could not solve, whereas BFSS solved 14 benchmarks that could not be solved by Cadet. Figure 3(b) gives the performance of the two algorithms with respect to time on the Arithmetic, Factorization and DisjDecomposition benchmarks. From the figure, we can see that while Cadet solves more benchmarks in

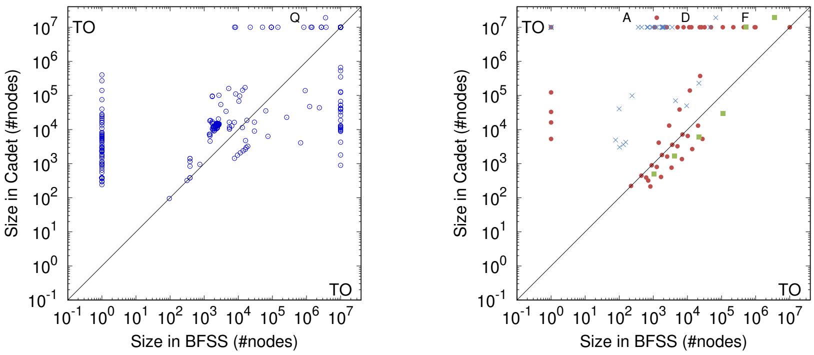

Fig. 4: BFSS vs CADET: Maximum Sizes of Skolem Functions. Legend: Please see Figure 1.

the QBFEVAL suite of benchmarks, BFSS solves more benchmarks than CADET in ARITHMETIC, DISJDECOMPOSITION and FACTORIZATION. In fact, in these categories, there were a total of 77 benchmarks that BFSS solved that CADET could not solve. Futhermore there was no benchmark in these suites that CADET could solve but BFSS could not. While CADET takes less time on some ARITHMETIC and QBFEVAL benchmarks, BFSS takes less time on DISJDECOMPOSITION and most FACTORIZATION benchmarks. Interestingly, most of the QBFEVAL benchmarks for which CADET takes less time, are solved in less than a minute by both CADET and BFSS.

We next compare the maximum sizes of the Skolem functions generated by CADET and BFSS. Note that CADET requires the input specification to be in QDIMACS format, whereas BFSS works with a DAG representation of the input. We compare the maximum sizes of the generated Skolem functions, since a specification given in QDIMACS format typically contains many output variables introduced due to Tseitin encoding of a non-CNF specification. Since the size of Skolem functions of Tseitin variables are usually small, this skews the average size of the Skolem functions generated, when comparing a tool that works with a QDIMACS representation of the specification (viz. CADET) with one that works with a DAG representation of the specification (viz. BFSS). Here, we find that for many of the QBFEVAL and DISJDECOMPOSITION benchmarks, the maximum sizes of the Skolem functions generated by BFSS are indeed smaller than those generated by CADET. On many of the ARITHMETIC and FACTORIZATION benchmarks, however, the sizes are comparable. There are, of course, cases where the sizes of Skolem functions generated by CADET are smaller than those generated by BFSS.

BFSS *vs* PARSYN*:* Figure 5 shows the comparison of time taken by BFSS and PARSYN. PARSYN was successful on a total of 200 benchmarks, with 118 in QBFEVAL, 64 in DISJDECOMPOSITION, 15 in ARITHMETIC and 3 in FACTORIZATION. Across all domains, BFSS solved 119 benchmarks that PARSYN could not solve. From Figure 5, we can see that on *every* benchmark across all domains, BFSS takes less time than PARSYN. We next compare the average sizes of the Skolem functions generated by the two algorithms in Figure 6. Here too, we observe that for most benchmarks, the

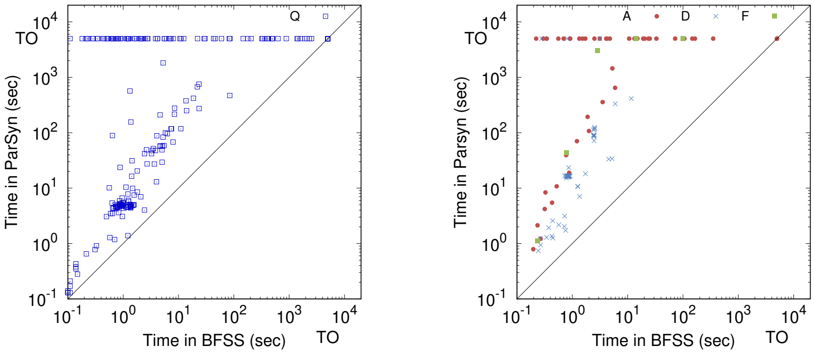

Fig. 5: BFSS vs PARSYN: Time Taken to synthesize Skolem Functions. Legend: Please see Figure 1.

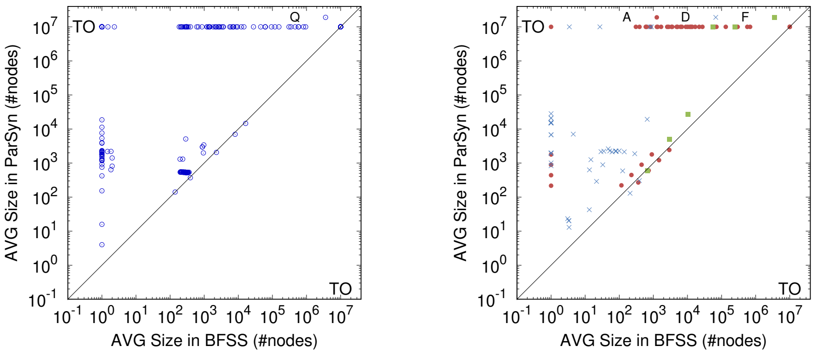

Fig. 6: BFSS vs PARSYN: Average Sizes of Skolem Functions. Legend: Please see Figure 1.

sizes of the Skolem functions generated by BFSS are smaller than those generated by PARSYN.

BFSS *vs* BAFSYN*:* We next compare the performance of BFSS with BAFSYN. BAFSYN was successful only on 11 benchmarks in QBFEVAL and could not solve any benchmark in the DISJDECOMPOSITION, ARITHMETIC and FACTORIZATION suites. However, BFSS was unable to solve the 11 benchmarks that BAFSYN solved. Similarly, none of 319 benchmarks solved by BFSS were solved by BAFSYN.

BFSS *vs* ABSSYNSKOLEM*:* ABSSYNSKOLEM was successful on 217 benchmarks, with 151 in QBFEVAL, 29 in DISJDECOMPOSITION, 32 in ARITHMETIC and 5 in FACTORIZATION suites. It could solve 19 benchmarks in QBFEVAL that BFSS could not solve. In contrast, BFSS solved 69 benchmarks in QBFEVAL, 39 in DISJDECOMPOSITION and 13 in ARITHMETIC – a total of 121 benchmarks – that ABSSYNSKOLEM could not solve. Figure 7 shows a comparison of running times of BFSS and ABSSYNSKOLEM. From the Figure we can see that BFSS takes less time than ABSSYNSKOLEM on most of the QBFEval, DisjDecomposition and Arithmetic benchmarks. AbsSynSkolem, however, takes less time on the Factorization benchmarks.

We next compare the average sizes of the Skolem functions generated by AbsSynSkolem and bfss in Figure 8. For QBFEval and DisjDecomposition, we found that the average size of Skolem Functions generated by AbsSynSkolem for most benchmarks was very small, and often close to 1. For many Arithmetic and Factorization benchmarks, the sizes generated by AbsSynSkolem were smaller than those generated by bfss.

In summary, bfss *(both pipelines considered together) outperforms all tools in the number of benchmarks that it could solve across all domains. In many instances, it takes less time and can solve instances that other tools have been unable to solve.*

7 Conclusion

In this paper, we showed complexity-theoretic hardness results for the Boolean functional synthesis problem. We then developed a two-phase approach to solve

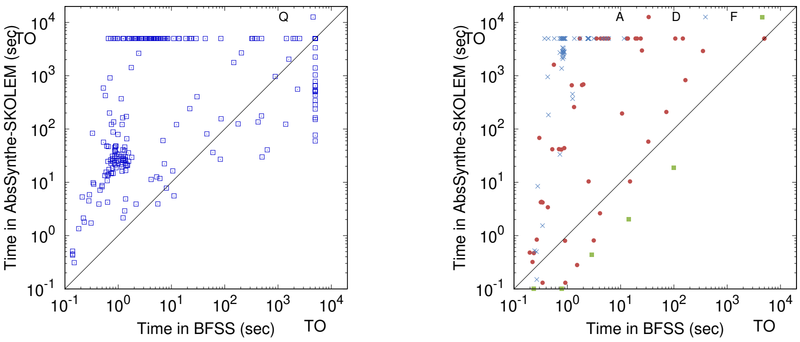

Fig. 7: bfss vs AbsSynSkolem: Time Taken to synthesize Skolem Functions. Legend: Please see Figure 1

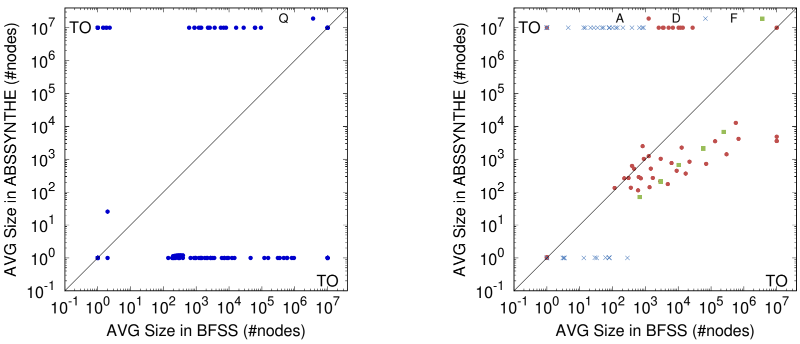

Fig. 8: bfss vs AbsSynSkolem: Average Sizes of Skolem Functions. Legend: Please see Figure 1.

this problem, where the first phase is an efficient algorithm that generates poly-sized functions and succeeds in solving a large number of benchmarks. For the remaining benchmarks, we employed the second phase of the algorithm that uses a expansion-based approach and builds Skolem functions by exploiting recent advances in SAT solvers. Extensive experiments show that our algorithm performs favourably over state-of-the-art tools when solving a large collection of benchmarks.

Acknowledgments

We thank Ajith K. John for many technical discussions. We also thank the anonymous reviewers for several pertinent remarks and suggestions.

References

1. Akshay S, Chakraborty S, John AK, Shah S (2017) Towards parallel Boolean functional synthesis. In: Proceedings of International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I, pp 337–353
2. Akshay S, Arora J, Chakraborty S, Krishna S, Raghunathan D, Shah S (2019) Knowledge compilation for Boolean functional synthesis. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp 161–169
3. Akshay S, Chakraborty S, Goel S, Kulal S, Shah S (2020) Code and benchmark details for BFSS experiments. `https://github.com/BooleanFunctionalSynthesis/bfss`
4. Alur R, Madhusudan P, Nam W (2005) Symbolic computational techniques for solving games. International Journal on Software Tools for Technology Transfer 7(2):118–128
5. Andersson G, Bjesse P, Cook B, Hanna Z (2002) A proof engine approach to solving combinational design automation problems. In: Proceedings of Design Automation Conference (DAC), pp 725–730
6. Baader F (1998) On the complexity of Boolean unification. Information Processing Letters 67:215–220
7. Balabanov V, Jiang JHR (2012) Unified QBF certification and its applications. Formal Methods in System Design 41(1):45–65
8. Boole G (1847) The Mathematical Analysis of Logic. Philosophical Library
9. Boudet A, Jouannaud JP, Schmidt-Schauss M (1989) Unification in Boolean rings and Abelian groups. Journal of Symbolic Computation 8(5):449–477
10. Brayton R, Mishchenko A (2010) ABC: An academic industrial-strength verification tool. In: Proceedings of International Conference on Computer-Aided Verification (CAV), pp 24–40
11. Brenguier R, Pérez GA, Raskin JF, Sankur O (2014) AbsSynthe: Abstract synthesis from succinct safety specifications. In: Proceedings of Workshop on Synthesis (SYNT), Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol 157, pp 100–116

12. Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 35(8):677–691
13. Chakraborty S, Fremont DJ, Meel KS, Seshia SA, Vardi MY (2015) On parallel scalable uniform SAT witness generation. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp 304–319
14. Chakraborty S, Fried D, Tabajara LM, Vardi MY (2018) Functional synthesis via input-output separation. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp 1–9
15. Chandrasekaran V, Srebro N, Harsha P (2008) Complexity of inference in graphical models. In: Proceedings of International Conference on Uncertainty in Artificial Intelligence (UAI), pp 70–78
16. Chen Y, Eickmeyer K, Flum J (2012) The Exponential Time Hypothesis and the parameterized clique problem. In: Proceedings of International Conference on Parameterized and Exact Computation (IPEC), pp 13–24
17. Darwiche A (2001) Decomposable negation normal form. Journal of the ACM 48(4):608–647
18. Deschamps JP (1972) Parametric solutions of Boolean equations. Discrete Mathematics 3(4):333–342
19. Fried D, Tabajara LM, Vardi MY (2016) BDD-based Boolean functional synthesis. In: Proceedings (Part II) of International Conference on Computer-Aided Verification (CAV), pp 402–421
20. Ganian R, Hlinený P, Langer A, Obdrzálek J, Rossmanith P, Sikdar S (2014) Lower bounds on the complexity of MSO_1 model-checking. Journal of Computer and System Sciences 80(1):180–194
21. Golia P, Roy S, Meel KS (2020) Manthan: A data-driven approach for Boolean function synthesis. In: Proceedings of International Conference on Computer-Aided Verification (CAV), pp 611–633
22. Hellerman L (1963) A catalog of three-variable Or-Invert and And-Invert logical circuits. IEEE Transactions on Electronic Computers EC-12(3):198–223
23. Heule M, Seidl M, Biere A (2014) Efficient Extraction of Skolem Functions from QRAT Proofs. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp 107–114
24. Impagliazzo R, Paturi R (2001) On the complexity of k-SAT. Journal of Computer and System Sciences 62(2):367–375
25. Jiang JHR (2009) Quantifier elimination via functional composition. In: Proceedings of International Conference on Computer-Aided Verification (CAV), Springer, pp 383–397
26. Jiang JHR, Lin HP, Hung WL (2009) Interpolating functions from large Boolean relations. In: Proceedings of International Conference on Computer-Aided Design (ICCAD), pp 779–784
27. Jo S, Matsumoto T, Fujita M (2012) SAT-based automatic rectification and debugging of combinational circuits with LUT insertions. In: Proceedings of Asian Test Symposium (ATS), pp 19–24
28. John A, Shah S, Chakraborty S, Trivedi A, Akshay S (2015) Skolem functions for factored formulas. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp 73–80
29. Karp R, Lipton R (1982) Turing machines that take advice. L'Enseignment Mathématique 28(2):191–209

30. Kuehlmann A, Krohm F (1997) Equivalence checking using cuts and heaps. In: Proceedings of Design Automation Conference (DAC), pp 263–268
31. Kuncak V, Mayer M, Piskac R, Suter P (2010) Complete functional synthesis. ACM SIGPLAN Notices 45(6):316–329
32. Löwenheim L (1910) Über die Auflösung von Gleichungen in Logischen Gebietkalkul. Mathematische Annalen 68:169–207
33. Macii E, Odasso G, Poncino M (1998) Comparing different Boolean unification algorithms. In: Conference Record of Asilomar Conference on Signals, Systems and Computers (Cat. No. 98CH36284), vol 2, pp 1052–1056
34. Martin U, Nipkow T (1989) Boolean unification - The story so far. Journal of Symbolic Computation 7(3-4):275–293
35. Niemetz A, Preiner M, Lonsing F, Seidl M, Biere A (2012) Resolution-based certificate extraction for QBF - (Tool presentation). In: Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), pp 430–435
36. QBFLib (2018) QBFEval 2018. `http://www.qbflib.org/qbfeval18.php`
37. Rabe MN (2019) Incremental determinization for quantifier elimination and functional synthesis. In: Proceedings of International Conference on Computer-Aided Verification (CAV), Part II, pp 84–94
38. Rabe MN, Seshia SA (2016) Incremental determinization. In: Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), pp 375–392
39. Rabe MN, Tentrup L (2015) CAQE: A certifying QBF solver. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp 136–143
40. Rabe MN, Tentrup L, Rasmussen C, Seshia SA (2018) Understanding and extending incremental determinization for 2QBF. In: Proceedings of International Conference on Computer-Aided Verification (CAV), Part II, pp 256–274
41. Silva JM, Lynce I, Malik S (2008) Conflict-driven clause learning SAT solvers. In: Biere A, Heule M, van Maaren H, Walsch T (eds) Handbook of Satisfiability, IOS Press, chap 14, pp 127–149
42. Solar-Lezama A (2013) Program sketching. International Journal on Software Tools for Technology Transfer 15(5-6):475–495
43. Solar-Lezama A, Rabbah RM, Bodík R, Ebcioglu K (2005) Programming by sketching for bit-streaming programs. In: Proceedings of International Conference on Programming Language Design and Implementation (PLDI), pp 281–294
44. Srivastava S, Gulwani S, Foster JS (2013) Template-based program verification and program synthesis. International Journal on Software Tools for Technology Transfer 15(5-6):497–518
45. Tabajara LM, Vardi MY (2017) Factored Boolean functional synthesis. In: Proceedings of International Confernce on Formal Methods in Computer-Aided Design (FMCAD), pp 124–131
46. Trivedi A (2003) Techniques in symbolic model checking. Master's thesis, Indian Institute of Technology Bombay, Mumbai, India
47. Zhu S, Tabajara LM, Li J, Pu G, Vardi MY (2017) Symbolic LTLf synthesis. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), pp 1362–1369