Witnessability of Undecidable Problems

SHUO DING, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA

Many problems in programming language theory and formal methods are undecidable, so they cannot be solved precisely. Practical techniques for dealing with undecidable problems are often based on decidable approximations. Undecidability implies that those approximations are always imprecise. Typically, practitioners use heuristics and *ad hoc* reasoning to identify imprecision issues and improve approximations, but there is a lack of computability-theoretic foundations about whether those efforts can succeed.

This paper shows a surprising interplay between undecidability and decidable approximations: there exists a class of undecidable problems, such that it is computable to transform any decidable approximation to a witness input demonstrating its imprecision. We call those undecidable problems *witnessable problems*. For example, if a program property P is witnessable, then there exists a computable function f_P, such that f_P takes as input the code of any program analyzer targeting P and produces an input program w on which the program analyzer is imprecise. An even more surprising fact is that the class of witnessable problems includes almost all undecidable problems in programming language theory and formal methods. Specifically, we prove the diagonal halting problem K is witnessable, and the class of witnessable problems is closed under complements and many-one reductions. In particular, all "non-trivial semantic properties of programs" mentioned in Rice's theorem are witnessable. We also explicitly construct a problem in the non-witnessable (and undecidable) class and show that both classes have cardinality $2^{\aleph_0}$.

Our results offer a new perspective on the understanding of undecidability: for witnessable problems, although it is impossible to solve them precisely, it is always possible to improve any decidable approximation to make it closer to the precise solution. This fact formally demonstrates that research efforts on such approximations are promising and shows there exist universal ways to identify precision issues of program analyzers, program verifiers, SMT solvers, etc., because their essences are decidable approximations of witnessable problems.

CCS Concepts: • **Theory of computation** → **Computability**; **Program reasoning**; **Automated reasoning**.

Additional Key Words and Phrases: Mathematical Logic, Program Semantics, Automated Theorem Proving

ACM Reference Format:
Shuo Ding and Qirun Zhang. 2023. Witnessability of Undecidable Problems. *Proc. ACM Program. Lang.* 7, POPL, Article 34 (January 2023), 21 pages. https://doi.org/10.1145/3571227

1 INTRODUCTION

Many problems in programming language theory and formal methods (program analysis [Landi 1992; Reps 2000], program verification [Abdulla and Jonsson 1996; Dima and Tiplea 2011], SMT solving [Bonacina et al. 2006; Day et al. 2018], type systems [Hu and Lhoták 2020; Pierce 1992; Wells 1999], etc.) consider complicated objects such as programs written in Turing-complete languages, and those problems are proved to be undecidable. It has been well-known since Turing and Church [Church 1936; Turing et al. 1936] that undecidable problems cannot be solved precisely.

Authors' addresses: Shuo Ding, Georgia Institute of Technology, Atlanta, USA, sding@gatech.edu; Qirun Zhang, Georgia Institute of Technology, Atlanta, USA, qrzhang@gatech.edu.

2475-1421/2023/1-ART34
https://doi.org/10.1145/3571227

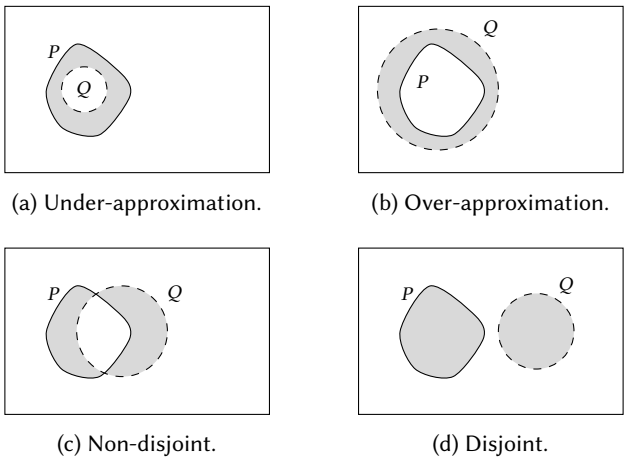
In practice, perhaps the best-known technique for handling undecidable problems is utilizing decidable approximations [Cousot and Cousot 1977; Hu and Lhoták 2020; Kildall 1973]. Undecidability implies that all approximations must be imprecise on infinitely many inputs—although theoretically important, it is a relatively discouraging result, and it does not shed light on improving the approximations encountered in practice.

This paper goes beyond previous work by presenting a surprising interplay between undecidability and decidable approximations. Specifically, we show that for a large class of undecidable problems, there exist *computable functions* that take as input the implementation (source code) of a decidable approximation and output a witness on which the approximation is imprecise. At first glance, this result appears counter-intuitive because, due to the nature of undecidability, for any arbitrarily given input, there is no general way to tell whether the approximation is imprecise on it. Otherwise, the problem would be decidable. Our result shows that there exists an algorithm that can compute imprecise inputs from the approximations. Our result does not aim to decide whether the approximations are imprecise on arbitrary inputs, thus bypassing the undecidability. Furthermore, this enables an iterative process: after computing a witness and improving the existing approximation, our result shows that we can always obtain a new witness, *i.e.*, the iterative process leads to more and more precise approximations. Note that this is fundamentally different from the idea of CounterExample-Guided Abstraction Refinement (CEGAR) [Clarke et al. 2000]: (1) in program verification, CEGAR often refines abstractions for each input program while we refine the program verifier itself; and (2) even if we apply the idea of CEGAR to refine a sound[1] program verifier when the verifier fails to prove the correctness of a program, it is, in general, undecidable to know whether that is a false positive. On the contrary, our approach directly constructs correct programs that cannot be proved correct by the verifier.

We state and prove our results using the terminology of computability theory. In the literature, computability theory has been primarily discussed using formal languages [Sipser 1996] and sets of natural numbers [Asperti 2008; Cutland 1980], and the two approaches are equivalent. Our work adopts the second approach: undecidable problems and their decidable approximations are both modeled as sets of natural numbers. Our result is general and applies to any Turing-complete programming language. In particular, natural numbers can encode any finite amount of information by computable encodings [Liang et al. 2014; Smorynski 1977]. Consider an undecidable problem P and its decidable approximations Q in Figure 1. Different approximation abstractions can lead to several set relationships between P and Q: Q is a subset of P (Figure 1a), Q is a superset of P (Figure 1b), Q intersects with P but is neither a subset nor a superset of P (Figure 1c), Q is disjoint from P (Figure 1d). To discuss all cases uniformly, we call the *symmetric difference* $P \triangle Q = (P \setminus Q) \cup (Q \setminus P)$ the *imprecision* of the approximation. For example, if Q is an under-approximation of P, then $P \triangle Q = P \setminus Q$, which represents the area of P not covered by Q. Utilizing the symmetric difference $P \triangle Q$ makes our results more general because the approximations are not restricted to under-approximations (Figure 1a) or over-approximations (Figure 1b).

We say an undecidable problem P is *witnessable* if and only if there exists a partial computable function w_P only depending on P, such that for any decidable approximation Q and its characteristic function[2] ϕ_q (the program implementing ϕ_q is encoded as a natural number q), $w_P(q)$ is defined (denoted as $w_P(q)\downarrow$) and $w_P(q) \in P \triangle Q$. Therefore, $w_P(q)$ is an *imprecision witness* of Q and w_P is a *witness function* of P. Our definition resembles the definition of productive sets in computability

[1] "Sound" means if the verifier concludes the program is correct, then the program is indeed correct. In other words, the verifier forms an *under-approximation* of the set of correct programs, which is typically implemented by *over-approximating* programs' behaviors (thus rejecting some correct programs). This convention is used throughout this paper.
[2] Recall that a set S's characteristic function is a 0-1 valued function f such that $f(x) = 1$ if and only if $x \in S$.



(a) Under-approximation. (b) Over-approximation.

(c) Non-disjoint. (d) Disjoint.

Fig. 1. Four cases of an undecidable problem P and its decidable approximation Q. The grey areas ($P \triangle Q$) represent the imprecision. The trivial case of $Q = \emptyset$ could be classified as either case (a) or case (d). The rectangle surrounding each case represents the set of all natural numbers.

theory [Soare 1999], but we focus on decidable approximations and do not require the approximation Q to be a subset of P.

This paper proves the following main results.

(1) The diagonal halting problem[3] $K = \{i \mid \phi_i(i) \downarrow\}$ is witnessable (Theorem 2);
(2) If P is witnessable, then its complement P^c is also witnessable (Theorem 3);
(3) If P_1 is witnessable and P_1 is many-one reducible to P_2, P_2 is also witnessable (Theorem 4).

These facts show that witnessable problems cover many undecidable problems in programming language theory and related fields. In particular, all "non-trivial semantics properties of programs" mentioned in Rice's theorem [Cutland 1980] and all "non-trivial complexity cliques" mentioned in intensional Rice's theorem [Asperti 2008] are witnessable. The satisfiability and validity of first-order logic formulas [Turing et al. 1936] and the Post correspondence problem [Post 1946] are also witnessable. Witnessability cannot be achieved via simple enumeration: in Figure 1b, by naively enumerating all programs and checking each of them using the characteristic function of Q, we are only guaranteed to find programs in Q or Q^c. But for programs in Q, we may never know whether they are in the symmetric difference area $P \triangle Q$ (marked as grey in Figure 1) because P is undecidable. Our approach, however, can directly compute a program that belongs to $P \triangle Q$.

The implications of our result are threefold.

(1) It shows the existence of universal ways to identify the precision issues of many algorithms in programming language theory and formal methods, including but not limited to program analyzers, program verifiers, SMT solvers, etc. In particular, the only restriction is that the algorithms should be total (*i.e.*, they terminate on every input).
(2) It shows common undecidable problems encountered in programming language theory and formal methods are more "tangible" than the folklore intuition of "being impossible to solve." In particular, although they are undecidable, the process of improving any given decidable approximation is computable. This provides a theoretical foundation for justifying why research efforts targeting those problems are promising and work well in practice.

[3]The diagonal halting problem and the traditional halting problem [Sipser 1996] are many-one reducible to each other.

(3) Many mathematical methods used in undecidability proofs are commonly regarded as ways to prove negative results (*e.g.*, undecidability). However, our results show that they also give ways to improve any given decidable approximation (thus, they also have positive effects).

The rest of this paper is organized as follows. Section 2 explains our basic notations and reviews computability theory. Section 3 gives our main results about witnessable problems. Section 4 explicitly constructs a non-witnessable problem. Section 5 shows the cardinalities of the two classes of problems. Section 6 uses two examples (program analyzers and SMT solvers) to discuss the implications for programming language theory and related fields. Section 7 presents discussions. Section 8 surveys related work. Section 9 concludes and discusses future research directions.

2 PRELIMINARY

Section 2.1 reviews basic set theory notations, Section 2.2 reviews computability theory, and Section 2.3 introduces our definition of decidable approximations.

2.1 Set Theory

To formulate computability theory concepts, we adopt the standard set theory notations [Jech 2013]: the "belongs to" relation $\in$, the strict subset relation $\subset$, the non-strict subset relation $\subseteq$, the set difference operator $\setminus$, and the set symmetric difference operator $\triangle$. In addition to that, we use $\mathbb{N}$ to denote the set of all natural numbers. Upper case letters $X, Y, Z \cdots$ denote subsets of $\mathbb{N}$, and lower case letters $x, y, z \cdots$ denote natural numbers. Given a set $S \subseteq \mathbb{N}$, the complement of S with respect to $\mathbb{N}$ is denoted as S^c, the power set of S is denoted as $\mathcal{P}(S)$, and the characteristic function χ_S of S is a function from $\mathbb{N}$ to the set $\{0, 1\}$ defined as follows:

$$\chi_S(x) = \begin{cases} 1 & \text{if} \quad x \in S \\ 0 & \text{if} \quad x \notin S. \end{cases}$$

2.2 Computability Theory

Two common ways to formulate computability theory are formal languages [Sipser 1996] and sets of natural numbers [Asperti 2008; Cutland 1980]. We follow the second approach: studying the (possibly relative) computability of subsets of $\mathbb{N}$ and partial functions from $\mathbb{N}$ to $\mathbb{N}$.

2.2.1 Partial Computable Functions. We use the standard notion of k-ary partial computable functions (from $\mathbb{N}^k$ to $\mathbb{N}$, $1 \leq k < +\infty$), which are partial functions computable by Turing machines, lambda calculus, or any equivalent models of computation [Cutland 1980; Sipser 1996]. The default arity of a partial computable function is one if it is not specified, and we mainly focus on 1-ary partial computable functions in this paper. In general, discussing 1-ary functions in computability theory suffices because a k-tuple of natural numbers can be computably converted to a single natural number and also be computably converted back (*e.g.*, Gödel's encoding [Smorynski 1977]). Given a partial computable function ϕ and a specific input $\vec{x}$, the notation "$\phi(\vec{x})\downarrow$" means that ϕ is defined on $\vec{x}$ and the notation "$\phi(\vec{x})\uparrow$" means that ϕ is undefined (or divergent) on $\vec{x}$. As an analogy, the undefined case of partial computable functions corresponds to the error state or the non-termination state of computer programs on a specific input. A partial computable function is total if and only if it is defined on every input. The domain of a partial computable function ϕ is the set of inputs on which ϕ is defined: $\{\vec{x} \mid \phi(\vec{x})\downarrow\}$. If two partial computable functions f and g have the same domain and output the same value on every input from the domain, then $f = g$ because they are the same partial computable function.

2.2.2 Numberings. For each arity k, we fix an admissible numbering [Rogers 1958] (enumeration) of k-ary partial computable functions (*e.g.*, the one generated by Kleene's normal form

theorem [Cutland 1980], or the one corresponding to an enumeration of all Turing machines):

$$\phi_0^k, \phi_1^k, \phi_2^k, \cdots,$$

where each $\phi_i^k, i \in \mathbb{N}$ is a partial computable function from $\mathbb{N}^k$ to $\mathbb{N}$. If $k = 1$ we write $\phi_i, i \in \mathbb{N}$. An index i in the numbering is analogous to a "program (code)" implementing the corresponding partial computable function, and $\Phi(i, x) = \phi_i(x)$ is analogous to an interpreter for the corresponding programming language. Note that there are infinitely many different indices corresponding to any single partial computable function, resembling the fact that there are infinitely many ways to implement the same function in common programming languages. This fact holds for all admissible numberings according to Rogers' equivalence theorem [Rogers 1958].

2.2.3 Computational Problems. A (computational) *problem* is formulated as a subset of $\mathbb{N}$. For example, the diagonal halting problem $K = \{i \mid \phi_i(i) \downarrow\}$ is the set of natural numbers representing programs that halt on themselves. The two names "problem" and "set" (of natural numbers) are used interchangeably throughout this paper. A problem is decidable if and only if its characteristic function is a total computable function, where any index for the characteristic function is called a *decider* for the problem. Otherwise, the problem is undecidable. The following two facts are used in the proofs of our main results.

Fact 1. *Decidable sets are closed under complements, finite unions, finite intersections, and addition/removal of finitely many natural numbers. In particular, any finite set is decidable.*

Proof. Those operations can easily be implemented using any Turing-complete programming language. By the Church-Turing thesis, the proof is completed. □

Fact 2. *If P is an undecidable problem, then both P and P^c are infinite.*

Proof. If P or P^c is finite, then P is decidable by Fact 1, which is a contradiction. □

We use the standard definition [Cutland 1980] of computably enumerable (c.e.) problems (also called recursively enumerable (r.e.) problems). A problem is c.e. if and only if it is the domain of a partial computable function. A problem is co-c.e. if and only if its complement is c.e.

2.2.4 Many-One Reductions. We use many-one reductions [Cutland 1980] to propagate imprecision witnesses among different problems.

Definition 1 (Many-One Reductions). *A problem P is many-one reducible to another problem Q (written as $P \leq_m Q$) if and only if there exists a total computable function f such that the following holds. f is called a many-one reduction from P to Q.*

$$\forall x \in \mathbb{N}, x \in P \Leftrightarrow f(x) \in Q.$$

Intuitively, a set P is many-one reducible to a set Q shows that Q is harder than P. In the above definition, to decide whether a natural number x belongs to P, we can first apply f on x and then test whether $f(x)$ belongs to Q. Thus if we can decide Q, we can also decide P.

2.2.5 S-M-N Theorem. We extensively use the S-m-n theorem [Cutland 1980] in our proofs. This theorem is analogous to partial evaluation [Jones et al. 1993] in programming languages.

Theorem 1 (S-m-n). *For any $m, n \in \mathbb{N}$, there exists an $(m+1)$-ary total computable function ψ_n^m such that the following holds for all $i, x_1, \cdots, x_m, y_1, \cdots, y_n \in \mathbb{N}$.*

$$\phi^n_{\psi_n^m(i,x_1,\cdots,x_m)}(y_1, \cdots, y_n) = \phi_i^{m+n}(x_1, \cdots, x_m, y_1, \cdots, y_n).$$

ψ_n^m corresponds to partial evaluators in programming languages. A trivial implementation of ψ_n^m is wrapping the code of ϕ_i^{m+n} (which is i) using fixed values of the first m inputs.

2.3 Decidable Approximations

Decidable approximations are commonly used in program analysis, program verification, etc., to deal with undecidable problems. Technically, for an undecidable set P, any decidable set of natural numbers could be regarded as an approximation of P, and different approximations have different precision/guarantees (Figure 1).

Definition 2 (Decidable Approximations). *Given an undecidable set P, any decidable set $Q \subseteq \mathbb{N}$ is a decidable approximation of P. In addition, if $Q \subset P$, then Q is called a decidable under-approximation of P; if $P \subset Q$, then Q is called a decidable over-approximation of P.*

Fact 3. *If P is an undecidable set, Q is a decidable approximation of P, then $P \triangle Q$ is infinite.*

Proof. If $P \triangle Q$ is finite, because Q is decidable, P is also decidable by Fact 1, which contradicts the assumption that P is undecidable. □

3 WITNESSABLE PROBLEMS

This section formally defines witnessable problems and presents three main results in Section 3.1, Section 3.2, and Section 3.3, respectively. Section 3.4 demonstrates that we can iteratively compute infinitely many imprecision witnesses for each decidable approximation.

Definition 3. *We say an undecidable problem $P \in \mathcal{P}(\mathbb{N})$ is witnessable if and only if there exists a partial computable function w_P, such that for any decidable approximation $Q \subseteq \mathbb{N}$ and any natural number q such that $\phi_q = \chi_Q$, $w_P(q)$ is defined and $w_P(q) \in P \triangle Q$. The partial computable function w_P is called a witness function of P, and $w_P(q)$ is called an imprecision witness of Q.*

The definition of w_P is general. First, it only depends on the problem P and works on any "implementation" (index) q of any decidable approximation Q. Second, it does not require the witness function w_P to be total computable: it only requires that w_P is defined on all indices of characteristic functions of decidable sets. This definition gives us more flexibility to construct such witness functions. In this paper, however, all constructed witness functions are total computable functions. Third, it only requires the existence of w_P. Whether there exists computable functions mapping P to w_P depends on how P is represented. In particular, many undecidability proofs [Asperti 2008; Cutland 1980; Ganesh et al. 2012] rely on many-one reductions from the halting problem or its complement. If P is given together with such a many-one reduction, then we can directly construct w_P from the given reduction (Theorems 2, 3, and 4).

Another important observation is that we can decide whether the imprecision witness is a false positive or a false negative. Indeed, because Q is decidable, if $\chi_Q(w_P(q)) = 0$, then $w_P(q) \in P \setminus Q$; if $\chi_Q(w_P(q)) = 1$, then $w_P(q) \in Q \setminus P$. This observation enables iterative imprecision witness computation described in Section 3.4.

3.1 Diagonal Halting Problem is Witnessable

Our proof idea is inspired by reconsidering the classical undecidability proof [Cutland 1980; Sipser 1996] of the diagonal halting problem K based on diagonalization. Specifically, replacing the hypothetical decider for K with an actual decider for any of K's decidable approximation Q yields an index in $K \triangle Q$.

Theorem 2 (Witnessability of Halt). *$K = \{i \mid \phi_i(i) \downarrow\}$ is witnessable.*

Proof. It is well-known that K is undecidable [Cutland 1980; Sipser 1996]. For any decidable approximation $Q \subseteq \mathbb{N}$ and a natural number q such that ϕ_q is the characteristic function of Q, we

construct a 2-ary partial computable function f using the universal function (interpreter) for all 1-ary partial computable functions (which interprets q as ϕ_q):

$$f(q,x) = \begin{cases} \uparrow & \text{if} \quad \phi_q(x) = 1 \\ 0 & \text{if} \quad \phi_q(x) = 0. \end{cases}$$

Because f is a 2-ary partial computable function, there exists an index j such that $f(q,x) \simeq \phi_j^2(q,x)$ for all $q, x \in \mathbb{N}$. By the S-m-n theorem, there exists a 2-ary total computable function ψ such that $\phi_j^2(q,x) \simeq \phi_{\psi(j,q)}(x)$ for all $q, x \in \mathbb{N}$. Next, we claim that $\psi(j,q) \in K \triangle Q$ by case analysis. To demonstrate this, we show that $\psi(j,q) \notin K \cap Q$ and $\psi(j,q) \notin (K \cup Q)^c$ both by contradiction, and these two facts imply $\psi(j,q) \in K \triangle Q$.

(1) If $\psi(j,q) \in K \cap Q$, then naturally $\psi(j,q) \in Q$, so we have

$$\begin{array}{ll} & \phi_q(\psi(j,q)) = 1 \\ \Longrightarrow & f(q,\psi(j,q)) \uparrow \\ \Longrightarrow & \phi_j^2(q,\psi(j,q)) \uparrow \\ \Longrightarrow & \phi_{\psi(j,q)}(\psi(j,q)) \uparrow . \end{array}$$

However, because $\psi(j,q) \in K$, we have $\phi_{\psi(j,q)}(\psi(j,q)) \downarrow$, which is a contradiction.

(2) If $\psi(j,q) \in (K \cup Q)^c$, in particular $\psi(j,q) \notin Q$, so we have

$$\begin{array}{ll} & \phi_q(\psi(j,q)) = 0 \\ \Longrightarrow & f(q,\psi(j,q)) = 0 \\ \Longrightarrow & \phi_j^2(q,\psi(j,q)) = 0 \\ \Longrightarrow & \phi_{\psi(j,q)}(\psi(j,q)) = 0. \end{array}$$

But on the other hand, since $\psi(j,q) \notin K$, we have $\phi_{\psi(j,q)}(\psi(j,q)) \uparrow$, which is a contradiction.

Combining the above two facts, we conclude that $\psi(j,q) \in K \triangle Q$. The witness function can be set as $w_K(q) = \psi(j,q)$. In particular, this w_K is a total function. □

The above theorem shows K is witnessable by constructing a valid witness function w_K. However, there exists more than one witness function for K as discussed in Section 7.1. Moreover, we show in Sections 3.2 and 3.3 that the class of witnessable problems is closed under complements and many-one reductions. In that sense, K is the starting point for deriving witnessable problems, but this does not mean that K is the only such starting point.

3.2 Witnessability is Closed Under Complements

The proof expresses the witness function for the complement problem P^c using the witness function for the original problem P.

Theorem 3 (Complement Closure). *If an undecidable problem P is witnessable, then its complement P^c is also witnessable.*

Proof. Suppose P is witnessable, and then there exists a partial computable function w_P such that for any decidable set Q and any natural number q such that ϕ_q computes χ_Q, $w_P(q)$ is defined and $w_P(q) \in P \triangle Q$. Now consider P^c. For any decidable set R and any natural number r such that ϕ_r computes χ_R, consider the following 2-ary partial computable function:

$$g(r,x) = \begin{cases} 0 & \text{if} \quad \phi_r(x) = 1 \\ 1 & \text{if} \quad \phi_r(x) = 0. \end{cases}$$

Without loss of generality, assume that $g(r,x) \simeq \phi_l^2(r,x)$. By the S-m-n theorem, there exists a 2-ary total computable function ψ such that $\phi_l^2(r,x) \simeq \phi_{\psi(l,r)}(x)$ for all $r, x \in \mathbb{N}$. Clearly, $\phi_{\psi(l,r)}(x)$

is the characteristic function of R^c. Now consider $w_P(\psi(l,r))$: according to the definition of w_P, $w_P(\psi(l,r))$ is defined and $w_P(\psi(l,r)) \in P \triangle R^c = P^c \triangle R$. Thus, we can set $w_{P^c}(r) = w_P(\psi(l,r))$. □

3.3 Witnessability is Closed Under Many-One Reductions

Assume that $P_1 \leq_m P_2$ and P_1 has a witness function; we show that P_2 also has a witness function. The proof composes a decidable approximation Q_2 of P_2 with the reduction from P_1 to P_2; this gives a decidable approximation Q_1 of P_1. Because we assume P_1 has a witness function, we can use that witness function to compute an imprecision witness for Q_1, and finally convert it back to an imprecision witness for Q_2.

The main technique used in our proof of Theorem 4 shares the same insight with Myhill [1957]'s proof showing that if A is a creative set, $A \leq_m B$, and B is c.e., then B is creative. But Myhill [1957]'s proof only targets one set relation (by definition, the productive function only considers c.e. *subsets* of the creative set's complement), while our proof handles symmetric difference, which covers all possible relations between a witnessable problem and its decidable approximations.

Theorem 4 (Many-One Reduction Closure). *If an undecidable problem P_1 is witnessable, and $P_1 \leq_m P_2$, then P_2 is also witnessable.*

Proof. Because P_1 is witnessable, there exists a witness function w_{P_1} for P_1. With the assumption $P_1 \leq_m P_2$, let f be a many-one reduction (a total computable function) from P_1 to P_2. According to the definition of many-one reductions (Definition 1), $\forall x \in \mathbb{N}, x \in P_1 \Leftrightarrow f(x) \in P_2$. To prove that P_2 is witnessable, we show that there exists a witness function w_{P_2}, such that for any computable set Q_2 and any natural number q_2 such that $\phi_{q_2} = \chi_{Q_2}$, $w_{P_2}(q_2)$ is defined and $w_{P_2}(q_2) \in P_2 \triangle Q_2$. Consider the total computable function $g(x) = \phi_{q_2}(f(x))$. It is a total computable function with function values in $\{0,1\}$, so it is a characteristic function of some decidable set Q_1. We can construct the following partial computable function:

$$h(i_1, i_2, x) = \phi_{i_1}(\phi_{i_2}(x)).$$

Suppose the index of h is j. By the S-m-n theorem, there exists a 3-ary total computable function ψ such that $h(i_1, i_2, x) \simeq \phi_{\psi(j,i_1,i_2)}(x)$ for all $i_1, i_2, x \in \mathbb{N}$. Because f is known, suppose it has an index k, and we have $g(x) \simeq h(q_2, k, x) \simeq \phi_{\psi(j,q_2,k)}(x)$. Thus, we obtained an index $q_1 = \psi(j, q_2, k)$ for χ_{Q_1}. By the definition of w_{P_1}, $w_{P_1}(q_1) \in P_1 \triangle Q_1$. Now we claim that $f(w_{P_1}(q_1)) \in P_2 \triangle Q_2$.

(1) If $f(w_{P_1}(q_1)) \in P_2 \cap Q_2$, then naturally we also have $f(w_{P_1}(q_1)) \in Q_2$, so $w_{P_1}(q_1) \in f^{-1}(Q_2) = Q_1$. But on the other hand, because $f(w_{P_1}(q_1)) \in P_2$, according to the definition of f, $w_{P_1}(q_1) \in P_1$. Combining those two, we have $w_{P_1}(q_1) \in P_1 \cap Q_1$, which is a contradiction because $w_{P_1}(q_1) \in P_1 \triangle Q_1$.
(2) If $f(w_{P_1}(q_1)) \in (P_2 \cup Q_2)^c$, then we have $w_{P_1}(q_1) \notin f^{-1}(Q_2) = Q_1$. According to the definition of f, we also have $w_{P_1}(q_1) \notin P_1$. Combining those two, we have $w_{P_1}(q_1) \in (P_1 \cup Q_1)^c$, which contradicts the fact that $w_{P_1}(q_1) \in P_1 \triangle Q_1$.

Clearly, $w_{P_2}(x) = f(w_{P_1}(\psi(j,x,k)))$ is a witness function for P_2, so P_2 is witnessable. □

There are many undecidable problems that can be proved by many-one reductions from K. In particular, all non-trivial semantic properties of programs mentioned in Rice's theorem are many-one reducible from K [Cutland 1980], and thus they are all witnessable according to the above theorem. Formally, an *index set* I is a subset of $\mathbb{N}$ satisfying $\forall i \in I, \forall j \in \mathbb{N}, (\phi_i = \phi_j \implies j \in I)$. An index set I is non-trivial if and only if $I \neq \emptyset$ and $I \neq \mathbb{N}$.

Corollary 1. *All non-trivial index sets are witnessable.*

Proof. Since there exists a many-one reduction from K to any non-trivial index set [Cutland 1980], this corollary immediately follows from Theorems 2 and 4. □

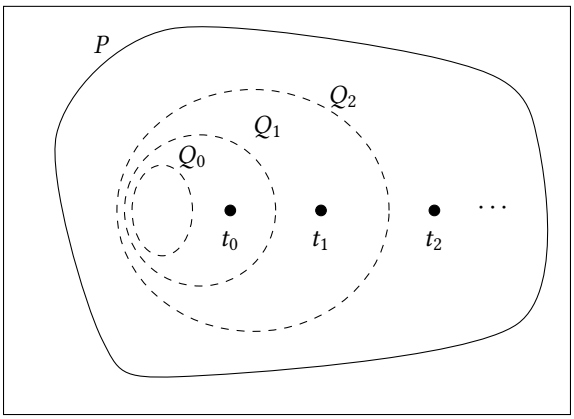


Fig. 2. The iterative imprecision witness computation. For an undecidable problem P, starting from a decidable under-approximation Q_0, after computing an imprecision witness t_0 for Q_0, we incorporate t_0 to get a better approximation Q_1, and compute an imprecision witness t_1 for Q_1, and so on. The big rectangle surrounding P represents the set of all natural numbers. Other kinds of approximations (Figure 1) are similar.

3.4 Iterative Imprecision Witness Computation

This section shows that we can compute infinitely many imprecision witnesses for each approximation of each witnessable problem: computing an imprecision witness, incorporating it into the approximation (in possibly naive ways), and repeating this process. Figure 2 illustrates this process based on under-approximations.

Note that a similar iterative process can also be done for productive sets and their c.e. subsets [Soare 1999], but in that case, the iterative process can only be done for subsets by definition, while our construction uses symmetric difference to handle more general set relations. This requires our observation "$P \setminus Q$ and $Q \setminus P$ can be distinguished by the computable set Q," so that we can determine whether the witness is a false positive or a false negative.

Theorem 5 (Iterative Witnesses). *If a problem P is witnessable and Q is a decidable approximation of P, then there is a 2-ary total computable function t such that for any ϕ_q computing χ_Q, $\{t(q,0), t(q,1), \ldots\}$ is an infinite list of different imprecision witnesses for Q.*

Proof. We describe how t works. First, $t(q,0)$ is defined as $w_P(q)$. Let $t_0 = t(q,0)$. We mainly discuss the case where $t_0 \in P \setminus Q$ and the other case ($t_0 \in Q \setminus P$) is similar. The two cases can be computably distinguished (because Q is decidable), so we can let t choose the correct case.

When $t_0 \in P \setminus Q$, we augment Q to obtain $Q' = Q \cup \{t_0\}$. The index for $\chi_{Q'}$ can be obtained using the following process. Consider the partial computable function g defined as follows:

$$g(y,z,x) = \begin{cases} 1 & \text{if} \quad x = z \\ \phi_y(x) & \text{if} \quad x \neq z. \end{cases}$$

Obviously, $h(x) = g(q, t_0, x)$ computes $\chi_{Q'}$. Suppose the index of g is j. By the S-m-n theorem we have a 3-ary total computable function ψ such that $g(y,z,x) \simeq \phi_{\psi(j,y,z)}(x)$ for all $y, z, x \in \mathbb{N}$. Thus, $\psi(j,q,t_0)$ is an index for $\chi_{Q'}$. If we apply the witness function w_P for P again on $\psi(j,q,t_0)$, then we have $w_P(\psi(j,q,t_0)) \in P \triangle Q'$, so $t_1 = w_P(\psi(j,q,t_0)) \neq t_0$.

When $t_0 \in Q \setminus P$, the set $Q \setminus \{t_0\}$ is also computable, and a similar construction of t_1 can be done.

We define $t(q,1)$ as t_1. For any $n \in \mathbb{N}$ and $n \geq 1$, repeating these computation steps n-times gives the definition of $t(q,n)$. □

The above theorem leads to an important implication: any witnessable problem must contain an infinite c.e. set because we can start from letting $Q = \emptyset$. This serves as a cornerstone for our construction of non-witnessable problems discussed in Section 4.

4 NON-WITNESSABLE PROBLEMS

Section 3 shows that witnessable problems include many undecidable problems. In this section, we construct an undecidable problem that is non-witnessable.

Lemma 1. *Given a witnessable problem $P \in \mathcal{P}(\mathbb{N})$, there exists a computably enumerable set $E \subseteq P$, such that both E and E^c are infinite.*

Proof. Suppose P is witnessable and let $Q = \emptyset$ be a decidable under-approximation of P. According to Theorem 5, we can compute infinitely many imprecision witnesses $t_0, t_1, t_2, \cdots \in P \triangle Q = P$. This list is clearly computably enumerable, and we let $E = \{t_0, t_1, t_2, \cdots\} \subseteq P$. Because P^c is infinite by Fact 2 and $E^c \supseteq P^c$, it is immediate that E^c is infinite. □

Based on the above lemma, we can construct an undecidable problem X not containing any c.e. set E such that both E and E^c are infinite. These sets are known as *immune sets* [Soare 1999]. Specifically, a set $I \subset N$ is immune if and only if I is infinite but does not contain any infinite c.e. set. Instead of directly adopting this definition, we provide our own construction to form a basis for proving Theorem 8 and make this paper more self-contained.

Our proof of Theorem 6 first lists all co-c.e. sets where both the sets and their complements are infinite (there are only countably many co-c.e. sets). Then, by a diagonalization-style construction, we can get a set X and prove that X is non-witnessable. The construction inherits some techniques of the construction in Post [1944]. However, Post [1944] constructs a simple set (*i.e.*, a c.e. set whose complement is immune), so its construction process needs to be c.e. On the contrary, we construct an immune set directly and do not require the construction process to be c.e.

Theorem 6 (Non-Witnessable Problem). *There exists a non-witnessable undecidable problem X.*

Proof. We list all co-c.e. sets $C_0, C_1, C_2, \cdots$ such that for all $i \in \mathbb{N}$, both C_i and C_i^c are infinite. It is easy to see that we can make this list $C_0, C_1, C_2, \cdots$ and also this list is infinite, because there are countably infinitely many c.e. sets such that both themselves and their complements are infinite. Now we construct a set Y by the following (infinite) process.

- Pick an arbitrary number y_0 from C_0^c, and include y_0 into Y.
- For each $i \in \mathbb{N}$ and $i \geq 1$, pick an arbitrary number y_i from C_i^c such that $y_i > y_{i-1} + 1$, and include y_i into Y.

This process is infinite because C_i^c is infinite for every $i \in \mathbb{N}$. Now let $X = Y^c$, and we claim that X is non-witnessable.

(1) It is easy to see that both X and Y are infinite, because $Y = \{y_0, y_1, y_2, \cdots\}$ is an infinite set and because we ensure that $y_i > y_{i-1} + 1$, the infinite set $\{y_0 + 1, y_1 + 1, y_2 + 1, \cdots\}$ is not included in Y.
(2) We then show that X is undecidable. Indeed, if X is decidable, then $Y = X^c$ is also decidable, and because decidable sets are co-c.e., we have $Y = C_j$ for some $j \in \mathbb{N}$. However, due to the construction, Y is different from every C_i (because we pick at least one element from C_i^c and include it into Y), which is a contradiction.
(3) Finally, we show that X does not contain any c.e. set E such that both E and E^c are infinite. Suppose there exists such an E, then $X \supseteq E$, $Y \subseteq E^c$, and E^c is an infinite co-c.e. set. Therefore, there exists a $j \in \mathbb{N}$ such that $Y \subseteq C_j$. However, due to the construction of Y, we know that Y contains at least one element from every C_i^c, which is a contradiction.

Lemma 1 claims that every witnessable problem must contain an infinite c.e. subset whose complement is also infinite, but X does not contain such a subset, so X cannot be witnessable. □

Explicitly constructing different kinds of "imprecision witnesses" is prevalent in mathematical logic. For example, Cantor's theorem states that for any set S, the cardinality of $\mathcal{P}(S)$ is strictly greater than the cardinality of S, and the typical proof is for any given injection f from S to $\mathcal{P}(S)$, we explicitly construct a set $\{x \in S \mid x \notin f(x)\}$ that is not in f's range [Jech 2013]. As another example, in the classical proof of Gödel's first incompleteness theorem [Smorynski 1977], for each formal system satisfying the conditions of that theorem, we can explicitly construct a "Gödel sentence" that is neither provable nor disprovable from the axioms of the formal system. In our case, witness functions only exist for witnessable problems.

5 CARDINALITIES OF THE TWO CLASSES OF PROBLEMS

The classes of witnessable problems and non-witnessable problems are both large. This section discusses the flexibility of constructing witnessable problems and non-witnessable problems and then proves that both classes of problems have cardinality $2^{\aleph_0}$.

We show that the class of witnessable problems has cardinality $2^{\aleph_0}$ based on the fact that this class is closed under many-one reductions (Theorem 4). Indeed, for the diagonal halting problem K, it is easy to construct a many-one reduction f from K to another problem such that $\mathbb{N} \setminus (f(K) \cup f(K^c))$ is infinite. The exact boundary between $f(K)$ and $f(K^c)$ can vary a lot: we can construct continuum many problems reducible from K. An immediate consequence is that many witness functions are shared by different witnessable problems, because the number of witness functions is countable.

Theorem 7 (Witnessable Class' Cardinality). *There are $2^{\aleph_0}$ witnessable problems.*

Proof. By Theorems 2 and 4, we know that any problem P such that $K \leq_m P$ is witnessable. We show that there are continuum many such problems. First, it is easy to construct a decidable set H such that H is infinite and $H \subset K$. This could be done, for example, by letting H be all Gödel numbers of terms that do not use the unbounded minimization operator. Pick a Gödel number $j \in K \setminus H$ (so ϕ_j is a total recursive function and the term corresponding to j uses the unbounded minimization operator) and consider the following total computable function:

$$f(x) = \begin{cases} j & \text{if} \quad x \in H \\ x & \text{else.} \end{cases}$$

It is easy to verify that $f(K) = K \setminus H$ and $f(K^c) = K^c$. Now, we can map the elements in the set $2^{\mathbb{N}}$ (whose elements are countably infinite sequences of 0's and 1's) to the subsets of H in a straightforward way, using an injection η from $2^{\mathbb{N}}$ to $\mathcal{P}(H)$. For each point s in $2^{\mathbb{N}}$, we obtain a unique witnessable set $f(K) \cup \eta(s)$, because $K \leq_m f(K) \cup \eta(s)$ by the above total computable function f. On the other hand, it is clear that there is an injection from the class of witnessable problems to $\mathcal{P}(\mathbb{N})$. Because both $\mathcal{P}(H)$ and $\mathcal{P}(\mathbb{N})$ have the cardinality $2^{\aleph_0}$, due to the Cantor-Bernstein theorem [Jech 2013], the cardinality of the class of witnessable problems is $2^{\aleph_0}$. □

The fact that there are continuum many non-witnessable problems is established by considering the diagonalization-style construction of X in Theorem 6: we are free to tweak the choice of each element in X so that we have two choices on each step. This gives continuum many versions of X.

Theorem 8 (Non-Witnessable Class' Cardinality). *There are $2^{\aleph_0}$ non-witnessable problems.*

Proof. Similar to the proof of Theorem 7, we only need to construct an injection from $2^{\mathbb{N}}$ to the class of non-witnessable problems. To this end, we generalize the construction of the set Y in Theorem 6 as follows.

- Pick two arbitrary numbers $y_{0,0}$ and $y_{0,1}$ from C_0^c, and include either $y_{0,0}$ or $y_{0,1}$ into Y.
- For each $i \in \mathbb{N}$ and $i \geq 1$, pick two arbitrary numbers $y_{i,0}$ and $y_{i,1}$ from C_i^c such that $\min(y_{i,0}, y_{i,1}) > \max(y_{i-1,0}, y_{i-1,1}) + 1$, and include either $y_{i,0}$ or $y_{i,1}$ into Y.

Because there are two choices for each step and there are countably infinitely many steps for constructing Y, we can easily correspond different elements in $2^{\mathbb{N}}$ with different constructed versions of Y, resulting in different versions of $X = Y^c$. This is indeed an injection from $2^{\mathbb{N}}$ to the class of non-witnessable problems, and it completes the proof. □

Because the two classes of undecidable problems have the same cardinality, we can regard these two classes as "having the same size." Because witnessable problems can be regarded as having a computable property (we can computably construct imprecision witnesses for any given approximation), this fact contrasts with decidable sets: the cardinality of the class of decidable sets is $\aleph_0$, which is strictly "smaller" than the cardinality of the class of undecidable sets ($2^{\aleph_0}$).

6 CASE STUDIES

This section presents two examples to demonstrate witness constructions using a simple programming language (defined in Section 6.1). Specifically, we convert the code of a sound sign analyzer to a program on which the analyzer is imprecise (Section 6.3), and convert the code of a sound string solver to a string formula on which the solver is imprecise (Section 6.4).

The constructions are not restricted to the two case studies. Moreover, the constructions are independent of the implementations of the program analyzers and SMT solvers. The analyzers and solvers do not need to be sound or complete. The only requirement is that they are written in Turing-complete programming languages and are total. In practice, both program analyzers and SMT solvers can be designed to run forever on certain cases, but it is easy to convert them to total programs by reporting "unknown" when their executions exceed certain resource limits.

Finally, the constructions discussed in this section do not intend to be the most cost-effective realizations to be used in practice, but show the theoretical possibility of computing such imprecision witnesses. Also, Section 7.1 shows that we can apply code optimizations on many steps during the construction, which creates more possible ways to realize this construction.

6.1 The Lang Programming Language

We discuss the construction based on a simple, but Turing-complete dynamically-typed programming language Lang defined in Figure 3. Lang resembles a very small subset of Racket [Flatt and PLT 2010]. Lang supports two basic types: (unbounded) integers and strings. Data structures definable in other programming languages can be encoded to (or decoded from) integers or strings.

The term ($\mathcal{F}$ e*) in Lang's definition represents all basic operations on basic types (such as integer arithmetic and comparisons). In the extreme case, we can stipulate $\mathcal{F}$ to represent all total computable functions, which is similar to Bruni et al. [2020]'s language definition.

We show our construction using Lang, but our construction is completely language-agnostic, *i.e.*, specific language features such as syntax, semantics, and type systems do not affect our construction as long as the language is Turing-complete.

6.2 Overview of Constructions

Figure 4 gives an overview of our case studies. Without loss of generality, we assume the analyzers and solvers are sound, meaning that they give under-approximations for the corresponding decision problems. Similar constructions can always be done for other kinds of approximations. Our construction consists of five steps based on our proofs of Theorems 2, 3, and 4.

(1) *Problem Construction (Figure 4a).* Construct a target decision problem D.

$$
\begin{array}{rcl}
\mathtt{var} & \in & \text{Var} \\
\mathtt{e} & ::= & \mathtt{var} \mid a \in \text{Int} \mid s \in \text{Str} \mid (\ \textbf{lambda}\ (\ \mathtt{var}*\ )\ \mathtt{e}\ ) \\
& \mid & (\ \mathcal{F}\ \mathtt{e}*\ ) \\
& \mid & (\ \textbf{letrec}\ (\ (\ \mathtt{var}\ \mathtt{e}\ )*\ )\ \mathtt{e}\ ) \\
& \mid & (\ \textbf{if}\ \mathtt{e}\ \mathtt{e}\ \mathtt{e}\ ) \\
& \mid & (\ \textbf{call}\ \mathtt{e}\ \mathtt{e}*\ )
\end{array}
$$

(a) Syntax of Lang. The notation "$*$" means the preceding symbol or parenthesized symbols occur zero or more times.

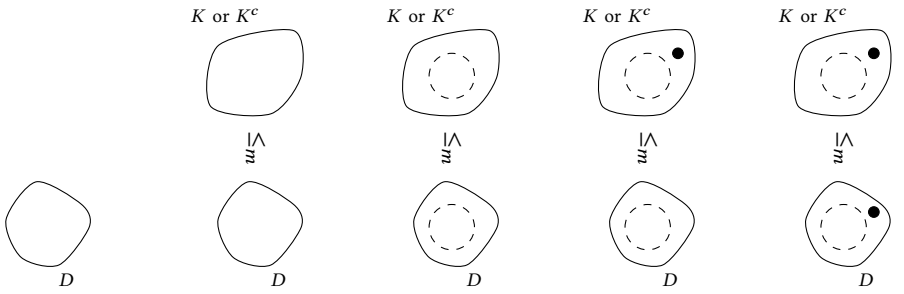
$$
\begin{array}{rcl}
[\![\text{env}, \mathtt{var}]\!] & = & \text{deref}[\text{env}[\mathtt{var}]] \\
[\![\text{env}, a]\!] & = & \text{a} \\
[\![\text{env}, s]\!] & = & \text{s} \\
[\![\text{env}, (\textbf{lambda}\ (\mathtt{var}*)\ \mathtt{e})]\!] & = & \langle \text{env}, \langle \mathtt{var}*, \mathtt{e} \rangle \rangle \\
[\![\text{env}, (\mathcal{F}\ \mathtt{e}*)]\!] & = & \mathcal{F}[[\![\text{env}, \mathtt{e}]\!]*] \\
[\![\text{env}, (\textbf{letrec}\ ((\mathtt{var}\ \mathtt{e}_1)*)\ \mathtt{e}_2)]\!] & = & [\![\text{env} + \text{bindrec}[\text{env}, \mathtt{var}*, \mathtt{e}_1*], \mathtt{e}_2]\!] \\
[\![\text{env}, (\textbf{if}\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3)]\!] & = & \text{ite}[[\![\text{env}, \mathtt{e}_1]\!], \text{env}, \mathtt{e}_2, \mathtt{e}_3] \\
[\![\text{env}, (\textbf{call}\ \mathtt{e}_1\ \mathtt{e}_2*)]\!] & = & [\![\text{env}_0 + \{(\mathtt{var} \mapsto \text{new}[[\![\text{env}, \mathtt{e}_2]\!]])*\}, \mathtt{e}_0]\!] \\
 & \text{where} & \langle \text{env}_0, \langle \mathtt{var}*, \mathtt{e}_0 \rangle \rangle = [\![\text{env}, \mathtt{e}_1]\!] \\
 & & \\
\text{bindrec}[\text{env}, \mathtt{var}*, \mathtt{e}*] & = & (\text{deref}[\text{env}_1[\mathtt{var}]] \leftarrow [\![\text{env} + \text{env}_1, \mathtt{e}]\!])*; \text{env}_1 \\
 & \text{where} & \text{env}_1 = \{(\mathtt{var} \mapsto \text{new}[\text{undefined}])*\} \\
\text{ite}[\text{true}, \text{env}, \mathtt{e}_1, \mathtt{e}_2] & = & [\![\text{env}, \mathtt{e}_1]\!] \\
\text{ite}[\text{false}, \text{env}, \mathtt{e}_1, \mathtt{e}_2] & = & [\![\text{env}, \mathtt{e}_2]\!] \\
\text{new}[\text{val}] & = & \text{Memory} \leftarrow \text{Memory} \cup \{\text{loc} \mapsto \text{val}\}; \text{loc} \\
\text{deref}[\text{loc}] & = & \text{Memory}[\text{loc}]
\end{array}
$$

(b) Semantics of Lang. $[\![\text{env}, \mathtt{e}]\!]$ means the evaluation result of the expression e in the environment env (mapping variables to memory locations). The order in env realizes variable shadowing. The initial environment is empty. $\langle\rangle$ means pairs. $a; b$ means sequencing (evaluating from left to right and returning the last value).

Fig. 3. Syntax and semantics of the simple programming language Lang. Any case not defined in the semantics is considered invalid where the program is treated as divergent (non-terminating).

(2) *Problem Reduction (Figure 4b).* Construct a many-one reduction from the diagonal halting problem K or its complement K^c to D.
(3) *Approximation Construction (Figure 4c).* Propagate the under-approximation of D to an under-approximation of K or K^c.
(4) *Witness Construction (Figure 4d).* Construct an imprecision witness in K or K^c.
(5) *Witness Mapping (Figure 4e).* Map the imprecision witness in K or K^c back to an imprecision witness in D.

Theorem 2 give the starting point for constructing witnesses, and Theorems 3 and 4 gives the flexibility to propagate witness constructions along complements and many-one reductions. Our construction steps follow these theorems. Overall, the construction steps specify an algorithm (the witness function) taking as input the under-approximating program for D (which is obtained by simply wrapping the code of the given program analyzer/SMT solver), and producing an output on which the original program analyzer/SMT solver is imprecise. Once the problem D and the



(a) Problem. (b) Reduction. (c) Approximation. (d) Witness. (e) Mapping back.

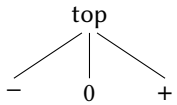
Fig. 4. Construction overview: (a) constructing a decision problem D, (b) constructing a many-one reduction $\leq_m$ from K to D, (c) constructing an approximation for K (the dashed circle in K) based on the approximation for D (the dashed circle in D), (d) constructing a witness in K (the black point in K), (e) mapping the witness in K back to a witness in D (the black point in D).

many-one reduction from K or K^c to D is fixed, this algorithm is also fixed, which is independent of any specific under-approximating program analyzers/SMT solvers.

6.3 Case Study 1: Program Analyzers

Given the code of a sound sign analyzer for Lang programs, we construct a program on which this analyzer is imprecise.

Analyzer Model. We stipulate that a sign analyzer takes as input a program and returns the following analysis results.



Specifically, "−," "0," and "+" denote that the input program, on every input, always halts and produces a negative integer, the integer 0, and a positive integer, respectively. The special value "top" indicates either the program does not satisfy any of the aforementioned cases or the sign analyzer is unable to determine the program's output sign.[4]

The Construction. Our construction requires a sound sign analyzer written in Lang for Lang programs and a Lang interpreter written in Lang. Their internal implementations are unconstrained. The sign analyzer is a (total) lambda taking an arbitrary Lang lambda (represented as a string) as input, and outputting "+," "−," "0," or "top."

```
1 (lambda (program)
2     (...code_of_analyzer...))
```

The Lang interpreter is a (partial) lambda that takes as input a single-parameter Lang lambda (represented as a string) with its input (also represented as a string), and returns the execution result of running the input lambda on the input (if it terminates). If the input is invalid[5] or a runtime error occurs during the interpretation, the interpreter enters an infinite loop.

```
1 (lambda (program, input)
```

[4]Practical program analyzers typically produce error messages on invalid programs. Our construction can wrap those analyzers so that they return "top" on invalid programs.

[5]The interpreter can also perform static type checking before the execution.

```
    (...code_of_interpreter...))
```

Step 1: Problem Construction. We convert the sign analysis problem to the problem D of verifying whether the input program returns a positive integer on every input. The sign analyzer can be converted to a program verifier for D, which is shown as follows. The return values of the verifier could be "correct" or "unknown." By the soundness of the sign analyzer, the verifier is an under-approximation of D. Note that "`=`" is one of the basic operations $\mathcal{F}$ in Lang's definition.

```
(lambda (program)
    (letrec ((analyzer
        (lambda (program)
            (...code_of_analyzer...))))
        (if (= (call analyzer program) "+")
            "correct"
            "unknown")))
```

Step 2: Problem Reduction. We construct a many-one reduction from the diagonal halting problem to the problem D constructed in Step 1. The reduction, assuming that the input is the code of a program p_1, returns the code of another program p_2 such that p_2 returns a positive integer on every input if and only if p_1 terminates on itself. The function `format` denotes filling placeholders (`[]`) in a string with extra string arguments (preserving quotes). For example, `(format "A[]C" "b")` evaluates to `"A\"b\"C"`. Note that `format` is one of the basic operations $\mathcal{F}$ in Lang's definition.

```
(lambda (program)
    (format
        "(lambda (input)
            (letrec ((interpreter
                (lambda (program, input)
                    (...code_of_interpreter...))))
                (if (call interpreter [] [])
                    1
                    0)))"
        program program))
```

Step 3: Approximation Construction. Using the reduction in Step 2, we convert the verifier in Step 1 to the following program q that under-approximates K.

```
(lambda (program)
    (letrec ((verifier (...code_of_verifier...))
        (reduction (...code_of_reduction...)))
        (if (= (call verifier (call reduction program)) "correct")
            "terminating"
            "unknown")))
```

Step 4: Witness Construction. Now we have the code of q that under-approximates K. Based on the proof of Theorem 2, we construct the following imprecision witness (program) witnessing the imprecision of q: the code of this program is in K, but q returns "unknown" on it.

```
(lambda (program)
    (letrec ((q (...code_of_q...)))
        (if (= (call q program) "terminating")
            (letrec ((loop (lambda () (call loop)))) (call loop))
            0)))
```

Step 5: Witness Mapping. According to the definition of the reduction in Step 2, the returned string of the following function call is an imprecision witness for the verifier that we constructed in Step 1, meaning that this witness terminates and returns a positive integer on every input, but the verifier returns "unknown" on it. As a result, the sign analyzer returns "top" on it.

```
(let ((reduction (...code_of_reduction...)))
    (call reduction "(...code_of_the_witness_for_K...)"))
```

6.4 Case Study 2: SMT Solvers

This section discusses our construction for a specific type of SMT solvers: string solvers. Specifically, we consider the validity problem of the set of sentences written as a $\forall\exists$ quantifier alternation applied to positive word equations described in Ganesh et al. [2012]'s work. For simplicity, we use $\mathbb{S}$ to denote the set of such sentences. Given the code of a sound solver for the validity problem of $\mathbb{S}$ sentences, we construct a valid $\mathbb{S}$ sentence on which the solver cannot conclude its validity.

Solver Model. We stipulate that a string solver takes an $\mathbb{S}$ sentence and returns either "valid," "invalid," or "unknown." Because the validity problem of $\mathbb{S}$ sentences is undecidable [Ganesh et al. 2012], any such solver must return "unknown" for some actually valid sentences.

The Construction. Our construction only requires a sound $\mathbb{S}$ solver written in Lang. The solver is a (total) lambda taking an $\mathbb{S}$ sentence (represented as a string) as input and outputting "valid," "invalid," or "unknown." If the input is not an $\mathbb{S}$ sentence, the solver should return "unknown."

```
(lambda (sentence)
    (...code_of_solver...))
```

Step 1: Problem Construction. The first step is to construct a decision problem D: the validity problem of $\mathbb{S}$ sentences. Because we assume the solver can return three different values, we wrap it into a lambda that returns only two values: "valid" or "unknown." By the soundness of the original solver, the wrapped solver results in an under-approximation of D.

```
(lambda (sentence)
    (letrec ((solver
        (lambda (sentence)
            (...code_of_solver...))))
        (if (= (call solver sentence) "valid")
            "valid"
            "unknown")))
```

Step 2: Problem Reduction. The second step is to construct a many-one reduction from K^c to the problem D constructed in Step 1. This step is based on a result due to Ganesh et al. [2012]. Specifically, given a two-counter machine M and a finite string w, Ganesh et al. [2012] construct an $\mathbb{S}$ sentence such that M does not halt on w if and only if this sentence is valid, and we denote this construction as f. For simplicity, we encode M and w into a single string Mw. On the other hand, because two-counter machines can simulate arbitrary Turing machines [Ganesh et al. 2012], we also have a computable function g such that for any pair of Lang program and input (L, v), both of which are represented as strings, $g((L, v)) = Mw$ is a string encoding a two-counter machine and its input such that M applied to w behaves the same as L applied to v. Finally, we construct the following reduction from K^c to D using f and g. Specifically, the input Lang program does not halt on itself if and only if the output is a valid $\mathbb{S}$ sentence.

```
(lambda (program)
    (letrec ((f (lambda (Mw) (...code_of_f...)))
        (g (lambda (L v) (...code_of_g...))))
        (call f (call g program program))))
```

Step 3: Approximation Construction. Using the reduction in Step 2, we convert the wrapped solver in Step 1 to the following program q that under-approximates K^c.

```
(lambda (program)
    (letrec ((wrapped_solver (...code_of_wrapped_solver...))
        (reduction (...code_of_reduction...)))
        (if (= (call wrapped_solver (call reduction program)) "valid")
            "non-terminating"
            "unknown")))
```

Step 4: Witness Construction. Now we have the code of q that under-approximates K^c. Based on the proofs of Theorem 2 and Theorem 3, we construct the following imprecision witness (program) witnessing the imprecision of q: its code is in K^c but q returns "unknown" for it.

```
(lambda (program)
    (letrec ((q (...code_of_q...)))
        (if (= (call q program) "non-terminating")
            0
            (letrec ((loop (lambda () (call loop)))) (call loop)))))
```

Step 5: Witness Mapping. According to the definition of the many-one reduction in Step 2, the returned string of the following function call is an imprecision witness for the wrapped solver that we constructed in Step 1, meaning that it is a valid $\mathbb{S}$ sentence but the wrapped solver returns "unknown" on it. As a result, the original solver also returns "unknown" on it.

```
(letrec ((reduction (...code_of_reduction...)))
    (call reduction "(...code_of_the_witness_for_Kc...)"))
```

7 DISCUSSIONS

7.1 The Flexibility of Constructing Imprecision Witnesses

In general, the construction of imprecision witnesses is flexible. For example, for a specific undecidable problem P such that $K \leq_m P$ (via the many-one reduction f), the construction of imprecision witnesses for a decidable approximation Q of P is parameterized by at least the following factors:

- Different programs (indices) of Q;
- Different programs (indices) of $f^{-1}(Q)$; and
- Different reductions f from K to P.

In particular, we can apply program optimizations on the program of Q and the program of $f^{-1}(Q)$. In addition, Theorem 5 states that we can perform iterative construction of imprecision witnesses indefinitely, where the method to "improve" the approximation at each step is also flexible.

Moreover, as mentioned in Section 3.1, K might not be the only starting point of the reduction. In particular, the construction of imprecision witnesses for K might be generalized to other diagonalization-based undecidability proofs, which we leave for future work.

7.2 The Classification of Undecidable Problems

Degree structures (*e.g.*, many-one degrees, and Turing degrees [Cutland 1980]) are commonly used to classify problems based on their relative computability. Our witnessable/non-witnessable problems, on the other hand, classify undecidable problems based on their "computable approximability": witnessable problems admit computable approximation improvements, while non-witnessable problems do not. We can still design decidable approximations for non-witnessable problems, but we do not have computable ways to automatically find imprecision witnesses.

We also compare our definition with several other classes of sets in computability theory. First, the class of witnessable problems is indeed different from the class of productive sets [Soare

1999] (despite the similarities between their definitions): productive sets cannot be recursively enumerable, but the recursively enumerable set K is witnessable. Second, it is easy to see that our class of witnessable problems is not contained in the class of all c.e. sets, because we prove that the cardinality of the class of witnessable problems is $2^{\aleph_0}$ while there are only countably many c.e. sets. One direct implication is that, in general, the infinite sequence of imprecision witnesses constructed in Theorem 5 may not cover all points in the undecidable problem being approximated (otherwise, the problem is c.e.). Third, our witnessable problems are different from the concept of limit computability. Indeed, the limit lemma [Shoenfield 1959] states that a problem P is limit computable if and only if $P \leq_T \emptyset'$, *i.e.*, P is Turing reducible to the first Turing jump of the empty set, where actually $\emptyset' = K$. Our class of witnessable problems includes the set of indices of all total functions $L = \{i \mid \forall x \in \mathbb{N}, \phi_i(x) \downarrow\}$ (because L is a non-trivial index set and thus is many-one reducible from K), and the Turing degree of L is $\mathbf{0}''$, so L is not Turing-reducible to $\emptyset'$.

7.3 Non-Witnessable Problems in Practice

Section 4 explicitly constructs a non-witnessable problem but does not relate that problem to any real scenarios in programming language theory and related fields. The spirit of that construction is similar to constructions in typical proofs of the time/space hierarchy theorems [Sipser 1996]: the constructed problems' main goal is to serve as a theoretical example to support the theorem instead of modeling any practical scenarios.

7.4 A Counter-Intuitive Fact: "Harder" Problems Do Not Prevent Witnessability

A counter-intuitive fact is that although many-one reduction is considered as a hardness comparison (*i.e.*, if $A \leq_m B$ then B is considered "harder" than A), accumulation of many-one reductions cannot make a witnessable problem hard enough to be non-witnessable. Imagine a finite but arbitrarily long chain of problems $P_0 <_m P_1 <_m P_2 <_m \cdots <_m P_n$, where $A <_m B$ is the strict version of $A \leq_m B$. Once we prove P_0 is witnessable, we know that P_n is still witnessable, despite that the many-one reductions show that P_n is much "harder" than P_0.

8 RELATED WORK

Extensive work exists on the undecidability of problems in programming language theory and related fields [Abdulla and Jonsson 1996; Bonacina et al. 2006; Day et al. 2018; Dima and Tiplea 2011; Hu and Lhoták 2020; Landi 1992; Pierce 1992; Reps 2000; Wells 1999]. Our work goes significantly beyond that: we analyze the "computable approximability" of different problems and provides computable imprecision witnesses for decidable approximations of certain undecidable problems.

There also exists work focusing on intensional aspects of computability results [Asperti 2008; Baldan et al. 2021; Moyen and Simonsen 2019]. Our result does not focus on extensional aspects or intensional aspects in particular, but rather on transforming the proofs of undecidability to witness functions. In other words, our result is applicable to both the traditional Rice's theorem [Cutland 1980] and some intensional versions of Rice's theorem [Asperti 2008].

Giacobazzi et al. [2015] and Bruni et al. [2020] propose constructions of incomplete cases for abstract interpretation, and abstract interpretation has been shown to be quite general to cover some other apparently different techniques [Cousot and Cousot 1995]. Our approach is even more general: we do not make any assumptions about what framework the program analyzer is based on (it could be based on abstract interpretation, but could also be based on arbitrary combinations of program analysis techniques [Aiken 1999; Cousot and Cousot 1977; Reps 1998] and arbitrary heuristics), and we do not require the program analyzer to be sound or complete.

In computability theory, the classes of problems that are similar to our class of witnessable problems include c.e. sets and limit computable sets, because they all describe certain kinds of

"approximating" processes. In Section 7.2, we discussed the difference between those two classes and our class, showing that our witnessable problems are indeed a new class of problems. Our classification motivation is also different from classifications based on relative computability with respect to oracles (such as Turing degrees and m-degrees): we classify undecidable problems based on decidable approximability.

Some of our proofs share similar ideas and methods with existing work. First, diagonalization and many-one reductions are standard techniques in computability [Cutland 1980], but we apply them to the scenario of our new concept (witnessability). Our proofs of Theorems 4, 5, and 6 share similar ideas with existing work in creative sets [Myhill 1957] and simple sets [Post 1944]. However, our work targets the new concept (witnessability) and more general set relations (modeled by the symmetric difference). The intent of our paper is not simply an extension of the existing work. Instead, our focus is witnessability's implications in programming language theory and formal methods, which shows that real (undecidable) problems and their approximations have the previously unknown "witness producing" computability property.

The word "approximation" is also used in algorithm design: for optimization problems, we can design algorithms whose outputs approximate the optimal solution [Vazirani 2013], and relevant approximability results are also developed [Arora 1998]. In contrast, our work focuses on decision problems instead of optimization problems, and we use decidable decision problems to approximate undecidable decision problems.

9 CONCLUSION AND FUTURE WORK

This paper defines witnessable problems, which are undecidable problems having computable imprecision witnesses for arbitrary decidable approximations. The class of witnessable problems has the same cardinality as the class of all undecidable problems. In particular, almost all problems in programming language theory and formal methods are witnessable, and algorithms in those areas are essentially decidable approximations of witnessable problems. Our results justify the research efforts on decidable approximations of witnessable problems and show the existence of universal ways to improve such approximations.

Witnessability is a newly developed concept that lies at the intersection of programming language theory and computability theory. We briefly outline some future directions. First, as discussed in Section 7.2, witnessable problems are different from many existing classes in computability theory, and it could be interesting future work to study more precise relations between witnessable problems and other known classes of problems [Cutland 1980; Myhill 1957; Post 1944; Soare 1999], as well as to study potential equivalent definitions of witnessable problems. Second, the definition of witness functions is very general, and in particular, we do not require witness functions to preserve semantic equivalence of programs (so given two input programs p_1 and p_2 computing the same function, a witness function w_P does not guarantee that $w_P(p_1)$ and $w_P(p_2)$ still compute the same function). It is thus interesting to study whether there always exist semantic-equivalence-preserving witness functions. Third, witnessability only concerns computability, but we can also consider extending it to involve complexity theory. Finally, we anticipate that the idea of the constructions proposed in this paper (Theorems 2, 3, 4, and 5) can shed light on more practical constructions that are useful in practice to improve decidable approximations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback on earlier drafts of this paper. We also thank Anton Bernshteyn for discussions about related topics in computability theory and descriptive set theory. This work was supported, in part, by Amazon under an Amazon Research Award in automated reasoning; by the United States National Science Foundation (NSF) under grants

No. 1917924 and No. 2114627; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

REFERENCES

Parosh Aziz Abdulla and Bengt Jonsson. 1996. Undecidable Verification Problems for Programs with Unreliable Channels. *Inf. Comput.* 130, 1 (1996), 71–90.

Alexander Aiken. 1999. Introduction to Set Constraint-Based Program Analysis. *Sci. Comput. Program.* 35, 2 (1999), 79–111.

Sanjeev Arora. 1998. The Approximability of NP-hard Problems. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*. ACM, 337–348.

Andrea Asperti. 2008. The intensional content of Rice's theorem. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 113–119.

Paolo Baldan, Francesco Ranzato, and Linpeng Zhang. 2021. A Rice's Theorem for Abstract Semantics. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference) (LIPIcs, Vol. 198)*, Nikhil Bansal, Emanuela Merelli, and James Worrell (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 117:1–117:19.

Maria Paola Bonacina, Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. 2006. Decidability and Undecidability Results for Nelson-Oppen and Rewrite-Based Decision Procedures. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4130)*. Springer, 513–527.

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. 2020. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.* 4, POPL (2020), 28:1–28:28.

Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (1936), 345–363.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1855)*. Springer, 154–169.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1995. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. ACM, 170–181.

Nigel Cutland. 1980. *Computability: An introduction to recursive function theory.* Cambridge university press.

Joel D. Day, Vijay Ganesh, Paul He, Florin Manea, and Dirk Nowotka. 2018. The Satisfiability of Word Equations: Decidable and Undecidable Theories. In *Reachability Problems - 12th International Conference, RP 2018, Marseille, France, September 24-26, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11123)*. Springer, 15–29.

Catalin Dima and Ferucio Laurentiu Tiplea. 2011. Model-checking ATL under Imperfect Information and Perfect Recall Semantics is Undecidable. *CoRR* abs/1102.4225 (2011).

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. 2012. Word Equations with Length Constraints: What's Decidable?. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7857)*, Armin Biere, Amir Nahir, and Tanja E. J. Vos (Eds.). Springer, 209–226.

Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 261–273.

Jason Z. S. Hu and Ondrej Lhoták. 2020. Undecidability of $d_{<:}$ and its decidable fragments. *Proc. ACM Program. Lang.* 4, POPL (2020), 9:1–9:30.

Thomas Jech. 2013. *Set theory.* Springer Science & Business Media.

Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Peter Sestoft.

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, Patrick C. Fischer and Jeffrey D. Ullman (Eds.). ACM Press, 194–206.

William Landi. 1992. Undecidability of Static Analysis. *LOPLAS* 1, 4 (1992), 323–337.

Shuying Liang, Weibin Sun, and Matthew Might. 2014. Fast Flow Analysis with Godel Hashes. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*. IEEE Computer Society, 225–234.

Jean-Yves Moyen and Jakob Grue Simonsen. 2019. More Intensional Versions of Rice's Theorem. In *Computing with Foresight and Industry - 15th Conference on Computability in Europe, CiE 2019, Durham, UK, July 15-19, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11558)*, Florin Manea, Barnaby Martin, Daniël Paulusma, and Giuseppe Primiero (Eds.). Springer, 217–229.

John Myhill. 1957. Creative sets. *Journal of Symbolic Logic* 22, 1 (1957).

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. ACM Press, 305–315.

Emil L Post. 1944. Recursively enumerable sets of positive integers and their decision problems. *Bull. Amer. Math. Soc.* 50, 5 (1944), 284–316.

Emil L Post. 1946. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.* 52, 4 (1946), 264–268.

Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726.

Thomas W. Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186.

Hartley Rogers. 1958. Gödel numberings of partial recursive functions. *The journal of symbolic logic* 23, 3 (1958), 331–341.

Joseph R Shoenfield. 1959. On degrees of unsolvability. *Annals of mathematics* (1959), 644–653.

Michael Sipser. 1996. Introduction to the Theory of Computation. *ACM Sigact News* 27, 1 (1996), 27–29.

Craig Smorynski. 1977. The incompleteness theorems. In *Studies in Logic and the Foundations of Mathematics*. Vol. 90. Elsevier, 821–865.

Robert I Soare. 1999. *Recursively enumerable sets and degrees: A study of computable functions and computably generated sets*. Springer Science & Business Media.

Alan Mathison Turing et al. 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* 58, 345-363 (1936), 5.

Vijay V Vazirani. 2013. *Approximation algorithms*. Springer Science & Business Media.

J. B. Wells. 1999. Typability and Type Checking in System F are Equivalent and Undecidable. *Ann. Pure Appl. Log.* 98, 1-3 (1999), 111–156.

Received 2022-07-07; accepted 2022-11-07