Decision Tree Learning in CEGIS-Based Termination Analysis

Satoshi Kura　　Hiroshi Unno　　Ichiro Hasuo

We present a novel decision tree-based synthesis algorithm of ranking functions for verifying program termination. Our algorithm is integrated into the workflow of CounterExample Guided Inductive Synthesis (CEGIS). CEGIS is an iterative learning model where, at each iteration, (1) a synthesizer synthesizes a candidate solution from the current examples, and (2) a validator accepts the candidate solution if it is correct, or rejects it providing counterexamples as part of the next examples. Our main novelty is in the design of a synthesizer: building on top of a usual decision tree learning algorithm, our algorithm detects *cycles* in a set of example transitions and uses them for refining decision trees. We have implemented the proposed method and obtained promising experimental results on existing benchmark sets of (non-)termination verification problems that require synthesis of piecewise-defined lexicographic affine ranking functions.

1 Introduction

Termination Verification by Ranking Functions and CEGIS

Termination verification is a fundamental but challenging problem in program analysis. Termination verification usually involves some well-foundedness arguments. Among them are those methods which synthesize *ranking functions* [16]: a ranking function assigns a natural number (or an ordinal, more generally) to each program state, in such a way that the assigned values strictly decrease along transition. Existence of such a ranking function witnesses termination, where well-foundedness of the set of natural numbers (or ordinals) is crucially used.

We study synthesis of ranking functions by CounterExample Guided Inductive Synthesis (CEGIS) [28]. CEGIS is an iterative learning model in which a synthesizer and a validator interact to find solutions for given constraints. At each iteration, (1) a synthesizer tries to find a candidate solution from the current examples, and (2) a validator accepts the candidate solution if it is correct, or rejects it providing counterexamples. These counterexamples are then used as part of the next examples (Fig. 1).

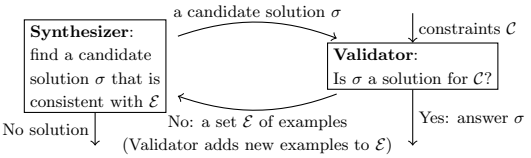


Fig. 1: the CEGIS architecture

CEGIS has been applied not only to program verification tasks (synthesis of inductive invariants [17, 18, 24, 25], that of ranking functions [19], etc.) but also to constraint solving (for CHC [12, 14, 27, 35], for pwCSP($\mathcal{T}$) [29,30], etc.). The success of CEGIS is attributed to the degree of freedom that synthesizers enjoy. In CEGIS, synthesizers receive a set of individual examples that synthesizers can use in various creative and speculative manners (such as machine learning). In contrast, in other methods such as [5–8,23,26], synthesizers receive logical constraints that are much more binding.

Segmented Synthesis in CEGIS-Based Ter-

内藏 理史　蓮尾 一郎, 総合研究大学院大学 & 国立情報学研究所, The Graduate University for Advanced Studies (SOKENDAI) & National Institute of Informatics.
海野 広志, 筑波大学 & 理化学研究所 革新知能統合研究センター, University of Tsukuba & RIKEN AIP.

mination Analysis

The choice of a *candidate space* for candidate solutions σ is important in CEGIS. A candidate space should be *expressive*: by limiting a candidate space, the CEGIS architecture may miss a genuine solution. At the same time, *complexity* should be low: a larger candidate space tends to be more expensive for synthesizers to handle.

This tradeoff is also in the choice of the type of examples: using an expressive example type, a small number of examples can prune a large portion of the candidate space; however, finding such expressive examples tends to be expensive.

In this paper, we use *piecewise affine functions* as our candidate space for ranking functions. Piecewise affine functions are functions of the form

$$f(\widetilde{x}) \quad = \quad \begin{cases} \widetilde{a}_1 \cdot \widetilde{x} + b_1 & \widetilde{x} \in L_1 \\ \quad \vdots & \\ \widetilde{a}_n \cdot \widetilde{x} + b_n & \widetilde{x} \in L_n \end{cases} \tag{1}$$

where $\{L_1, \ldots, L_n\}$ is a partition of the domain of $f(\widetilde{x})$ such that each L_i is a polyhedron (i.e. a conjunction of linear inequalities). We say *segmented synthesis* to emphasize that our synthesis targets are piecewise affine functions with case distinction. Piecewise affine functions stand on a good balance between expressiveness and complexity: the tasks of synthesizers and validators can be reduced to linear programming (LP); at the same time, case distinction allows them to model a variety of situations, especially where there are discontinuities in the function values and/or derivatives.

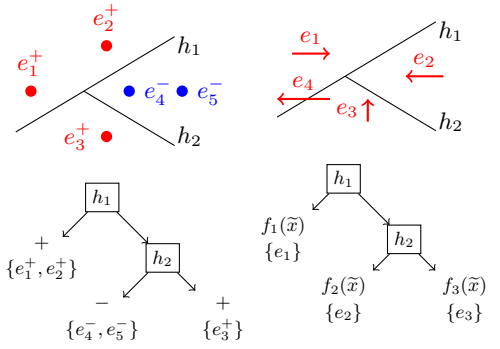
We use *transition examples* as our example type (Table 1). Transition examples are pairs of program states that represent transitions; they are much cheaper to handle compared to *trace examples* (finite traces of executions until termination) used e.g. in [15, 32]. The current work is the first to pursue segmented synthesis of ranking functions with transition examples; see Table 1.

Decision Tree Learning for CEGIS-Based Termination Analysis: a Challenge

In this paper, we represent piecewise affine functions (1) by the data structure of *decision trees*. The data structure suits the CEGIS architecture (Fig. 1): iterative refinement of candidate solutions can be naturally expressed by growing decision trees. The main challenge of this paper is the design of an effective synthesizer for decision trees—such a synthesizer *learns* decision trees from examples.

Table 1: ranking function synthesis by CEGIS

example type \ candidate space	trace examples	transition examples
affine ranking functions	[15, 32]	[19]
piecewise affine ranking functions	[15, 32]	our method



(a) For invariants (b) For ranking functions

Fig. 2: Decision tree learning

In fact, decision tree learning in the CEGIS architecture has already been actively pursued, for the synthesis of *invariants* as opposed to ranking functions [12,14,18,22,35]. It is therefore a natural idea to adapt the decision tree learning algorithms used there, from invariants to ranking functions. However, we find that a naive adaptation of those algorithms for invariants does not suffice: they are good at handling *state examples* that appear in CEGIS for invariants; but they are not good at handling transition examples.

More specifically, when decision tree learning is applied to invariant synthesis (Fig. 2a), examples are given in the form of program states labeled as positive or negative. Decision trees are then built by iteratively selecting the best halfspaces—where "best" is in terms of some quality measures—until each leaf contains examples with the same label. One common quality measure used here is an information-theoretic notion of *information gain*.

We extend this from invariant synthesis to ranking function synthesis where examples are given by

transitions instead of states (Fig. 2b). In this case, a major challenge is to cope with examples that cross a border of the current segmentation—such as the transition e_4 crossing the border h_1 in Fig. 2b. Our decision tree learning algorithm should handle such crossing examples, taking into account the constraints imposed on the leaf labels affected by those examples (the affected leaf labels are $f_1(\widetilde{x})$ and $f_3(\widetilde{x})$ in the case of e_4).

Our Algorithm: Cycle-Based Decision Tree Learning for Transition Examples

We use what we call the *cycle detection theorem* (Theorem 17) as a theoretical tool to handle such crossing examples. The theorem claims the following: if there is no piecewise affine ranking function with the current segmentation of the domain (such as the one in Fig. 2b given by h_1 and h_2), then this must be caused by a certain type of cycle of constraints, which we call an *implicit cycle*.

In our decision tree learning algorithm, when we do not find a piecewise affine ranking function with the current segmentation, we find an implicit cycle and refine the segmentation to break the cycle. Once all the implicit cycles are gone, the cycle detection theorem guarantees the existence of a candidate piecewise affine ranking function with the segmentation.

We integrate this decision tree learning algorithm in the CEGIS architecture (Fig. 1) and use it as a synthesizer. Our implementation of this framework gives promising experimental results on existing benchmark sets.

Contribution

Our contribution is summarized as follows.

- We provide a decision tree-based synthesizer for ranking functions integrated into the CEGIS architecture. Our synthesizer uses transition examples to find candidate piecewise affine ranking functions. A major challenge here, namely handling constraints arising from crossing examples, is coped with by our theoretical observation of the cycle detection theorem.
- We implement our synthesizer for ranking functions implemented in MuVal and report the experience of using MuVal for termination and non-termination analysis. The experiment results show that MuVal's performance is comparable to state-of-the-art termination analyzers [7, 10, 13, 21] from Termination Competition 2020, and that MuVal can prove (non-)termination of some benchmarks with which other analyzers struggle.

Organization.

Section 2 shows the overview of our method via examples. Section 3 explains our target class of predicate constraint satisfaction problems and how to encode (non-)termination problem into such constraints. In Section 4, we review CEGIS architecture, and then explain simplification of examples into positive/negative examples. Section 5 proposes our main contribution, our decision tree-based ranking function synthesizer. Section 6 shows our implementation and experimental results. Related work is discussed in Section 7, and we conclude in Section 8.

2 Preview by Examples

We present a preview of our method using concrete examples. We start with an overview of the general CEGIS architecture, after which we proceed to our main contribution, namely a decision tree learning algorithm for transition examples.

2.1 Termination Verification by CEGIS

Our method follows the usual workflow of termination verification by CEGIS. It works as follows: given a program, we encode the termination problem into a constraint solving problem, and then use the CEGIS architecture to solve the constraint solving problem.

Encoding the termination problem.

The first step of our method is to encode the termination problem as the set $\mathcal{C}$ of constraints.

Example 1. As a running example, consider the following C program.

```
while(x != 0) {
    if(x < 0) { x++; } else { x--; }
}
```

The termination problem is encoded as the following constraints.

$$x < 0 \land x' = x + 1 \implies R(x, x') \quad (2)$$

$$\neg(x < 0) \land x' = x - 1 \implies R(x, x'). \quad (3)$$

Here, R is a predicate variable representing a well-founded relation, and term variables x, x' are universally quantified implicitly.

The set $\mathcal{C}$ of constraints claims that the transition relation for the given program is subsumed by

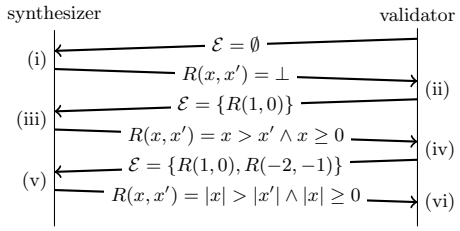


Fig. 3: An example of CEGIS iterations

a well-founded relation. So, verifying termination is now rephrased as the existence of a solution for $\mathcal{C}$. Note that we omitted constraints for invariants for simplicity in this example (see Sect. 3 for the full encoding).

Constraint solving by CEGIS.

The next step is to solve $\mathcal{C}$ by CEGIS.

In the CEGIS architecture, a synthesizer and a validator iteratively exchange a set $\mathcal{E}$ of examples and a candidate solution $R(x, x')$ for $\mathcal{C}$. At the moment, we present a rough sketch of CEGIS, leaving the details of our implementation to Sect. 2. 2.

Example 2. Fig. 3 shows how the CEGIS architecture solves the set $\mathcal{C}$ of constraints shown in (2) and (3). Fig. 3 consists of three pairs of interactions (i)-(vi) between a synthesizer and a validator.

(i) The synthesizer takes $\mathcal{E} = \emptyset$ as a set of examples and returns a candidate solution $R(x, x') = \bot$ synthesized from $\mathcal{E}$. In general, candidate solutions are required to satisfy all constraints in $\mathcal{E}$, but the requirement is vacuously true in this case.

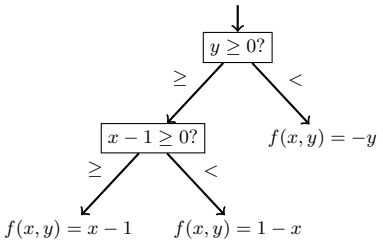
(ii) The validator receives the candidate solution and finds out that the candidate solution is not a genuine solution. The validator finds that the assignment $x = 1, x' = 0$ is a counterexample for (3), and thus adds $R(1, 0)$ to $\mathcal{E}$ to prevent the same candidate solution in the next iteration.

(iii) The synthesizer receives the updated set $\mathcal{E} = \{R(1,0)\}$ of examples, finds a ranking function $f(x) = x$ for $\mathcal{E}$ (i.e. for the transition from $x = 1$ to $x' = 0$), and returns a candidate solution $R(x, x') = x > x' \wedge x \geq 0$.

(iv) The validator checks the candidate solution, finds a counterexample $x = -2, x' = -1$ for (2), and adds $R(-2, -1)$ to $\mathcal{E}$.

(v) The synthesizer finds a ranking function $f(x) = |x|$ for $\mathcal{E}$ and returns $R(x, x') = |x| > |x'| \wedge |x| \geq 0$ as a candidate solution. Note that the synthesizer have to synthesize a piecewise affine function here, but details are deferred to Sect. 2. 2.

(vi) The validator accepts the candidate solution because it is a genuine solution for $\mathcal{C}$.



$$f(x,y) = \begin{cases} x-1 & y \geq 0 \wedge x-1 \geq 0 \\ 1-x & y \geq 0 \wedge x-1 < 0 \\ -y & y < 0 \end{cases}$$

Fig. 4: An example of a decision tree that represents a piecewise affine ranking function $f(x, y)$

2. 2 Handling Cycles in Decision Tree Learning

We explain the importance of handling cycles in our decision tree-based synthesizer of piecewise affine ranking functions.

In what follows, we deal with such decision trees as shown in Fig. 4: their internal nodes have affine inequalities (i.e. halfspaces); their leaves have affine functions; and overall, such a decision tree expresses a piecewise affine function (Fig. 4). When we remove leaf labels from such a decision tree, then we obtain a template of piecewise functions where condition guards are given but function bodies are not. We shall call the latter a *segmentation*.

Input and output of our synthesizer.

The input of our synthesizer is a set $\mathcal{E}$ of transition examples (e.g. $\mathcal{E} = \{R(1,0), R(-2,-1)\}$) as explained in Sect. 2. 1. The output of our synthesizer is a well-founded relation $R(\widetilde{x}, \widetilde{x}') \coloneqq f(\widetilde{x}) > f(\widetilde{x}') \wedge f(\widetilde{x}) \geq 0$ where $\widetilde{x}$ is a sequence of variables and $f(\widetilde{x})$ is a piecewise affine function, which is represented by a decision tree (Fig. 4). Therefore our synthesizer aims at *learning* a suitable decision tree.

Refining segmentations and handling cy-

cles.

Roughly speaking, our synthesizer learns decision trees in the following steps.

1. Generate a set H of halfspaces from the given set $\mathcal{E}$ of examples. This H serves as the vocabulary for internal nodes. Set the initial segmentation to be the one-node tree (i.e. the trivial segmentation).
2. Try to synthesize a piecewise affine ranking function f for $\mathcal{E}$ with the current segmentation—that is, try to find suitable leaf labels. If found, then use this f in a candidate well-founded relation $R(\widetilde{x}, \widetilde{x}') = f(\widetilde{x}) > f(\widetilde{x}') \wedge f(\widetilde{x}) \geq 0$.
3. Otherwise, refine the current segmentation with some halfspace in H, and go to Step 2.

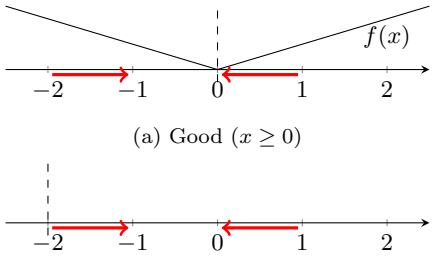
The key step of our synthesizer is Step 3. We show a few examples.

Example 3. Suppose we are given $\mathcal{E} = \{R(1,0), R(-2,-1)\}$ as a set of examples. Our synthesizer proceeds as follows: (1) Our synthesizer generates the set $H \coloneqq \{x \geq 1, x \geq 0, x \geq -2, x \geq -1\}$ from the examples in $\mathcal{E}$. (2) Our synthesizer tries to find a ranking function of the form $f(x) = ax + b$ (with the trivial segmentation), but there is no such ranking function. (3) Our synthesizer refines the current segmentation with $(x \geq 0) \in H$ because $x \geq 0$ "looks good". (4) Our synthesizer tries to find a ranking function of the form $f(x) =$ **if** $x \geq 0$ **then** $ax+b$ **else** $cx+d$, using the current segmentation. Our synthesizer obtains $f(x) =$ **if** $x \geq 0$ **then** x **else** $-x$ and use this $f(x)$ for a candidate solution.

How can we decide which halfspace in H "looks good"? We use *quality measure* that is a value representing the quality of each halfspace and select the halfspace with the maximum quality measure.

Fig. 5 shows the comparison of the quality of $x \geq 0$ and $x \geq -2$ in this example. Intuitively, $x \geq 0$ is better than $x \geq -2$ because we can obtain a simple ranking function **if** $x \geq 0$ **then** x **else** $-x$ with $x \geq 0$ (Fig. 5a) while we need further refinement of the segmentation with $x \geq -2$ (Fig. 5b). In Sect. 5, we introduce a quality measure for halfspaces following this intuition.

Our synthesizer iteratively refines segmentations following this quality measure, until examples contained in each leaf of the decision tree admit an affine ranking function. This approach is inspired by the use of information gain in the decision tree learning for invariant synthesis.



(a) Good ($x \geq 0$)

(b) Bad ($x \geq -2$)

Fig. 5: Selecting halfspaces. Transition examples are shown by red arrows. Boundaries of halfspaces are shown by dashed lines.

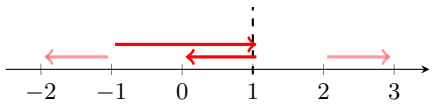


Fig. 6: Two examples $R(-1,1)$ and $R(1,0)$ make an implicit cycle between $x \geq 1$ and $\neg(x \geq 1)$.

Example 3 showed a natural extension of a decision tree learning method for invariant synthesis. However, this is not enough for transition examples, for the reasons of *explicit* and *implicit cycles*. Here are their examples.

Example 4. Suppose we are given $\mathcal{E} = \{R(1,0), R(0,1)\}$. In this case, there is no ranking function because $\mathcal{E}$ contains a cycle $1 \to 0 \to 1$ witnessing non-termination. We call such a cycle an *explicit cycle*.

Example 5. Let $\mathcal{E} = \{R(-1,1), R(1,0), R(-1,-2), R(2,3)\}$ (Fig. 6). Our synthesizer proceeds as follows. (1) Our synthesizer generates the set $H \coloneqq \{x \geq 1, x \geq 0, \dots\}$ of halfspaces. (2) Our synthesizer tries to find a ranking function of the form $f(x) = ax + b$ (with the trivial segmentation), but there is no such. (3) Our synthesizer refines the current segmentation with $(x \geq 1) \in H$ because $x \geq 1$ "looks good" (i.e. is the best with respect to a quality measure).

We have reached the point where the naive extension of decision tree learning explained in Example 3 no longer works: although all constraints contained in each leaf of the decision tree admit an affine ranking function, there is no piecewise affine

ranking function for $\mathcal{E}$ of the form $f(x) = \textbf{if } x \geq 1 \textbf{ then } ax + b \textbf{ else } cx + d$.

More specifically, in this example, the leaf representing $x \geq 1$ contains $R(2, 3)$, and the other leaf representing $\neg(x \geq 1)$ contains $R(-1, -2)$. The example $R(2, 3)$ admits an affine ranking function $f_1(x) = -x + 2$, and $R(-1, -2)$ admits $f_2(x) = x + 1$, respectively. However, the combination $f(x) = \textbf{if } x \geq 1 \textbf{ then } f_1(x) \textbf{ else } f_2(x)$ is not a ranking function for $\mathcal{E}$. Moreover, there is no ranking function for $\mathcal{E}$ of the form $f(x) = \textbf{if } x \geq 1 \textbf{ then } ax + b \textbf{ else } cx + d$.

It is clear that this failure is caused by the *crossing examples* $R(-1, 1)$ and $R(1, 0)$. It is not that every crossing example is harmful. However, in this case, the set $\{R(-1, 1), R(1, 0)\}$ forms a cycle between the leaf for $x \geq 1$ and the leaf for $\neg(x \geq 1)$ (see Fig. 6). This "cycle" among leaves—in contrast to *explicit* cycles such as $\{R(1, 0), R(0, 1)\}$ in Example 4—is called an *implicit cycle*.

Once an implicit cycle is found, our synthesizer cuts it by refining the current segmentation. Our synthesizer continues the above steps (1–3) of decision tree learning as follows. (4) Our synthesizer selects $(x \geq 0) \in H$ and cuts the implicit cycle $\{R(-1, 1), R(1, 0)\}$ by refining segmentations. (5) Using the refined segmentation, our synthesizer obtains $f(x) = \textbf{if } x \geq 1 \textbf{ then } -x + 2 \textbf{ else if } x \geq 0 \textbf{ then } 0 \textbf{ else } x + 3$ as a ranking function for $\mathcal{E}$.

As explained in Example 4,5, handling (explicit and implicit) cycles is crucial in decision tree learning for transition examples. Moreover, our *cycle detection theorem* (Theorem 17) claims that if there is no explicit or implicit cycle, then one can find a ranking function for $\mathcal{E}$ without further refinement of segmentations.

3 (Non-)Termination Verification as Constraint Solving

We explain how to encode (non-)termination verification to constraint solving.

Following [30], we formalize our target class pwCSP of predicate constraint satisfaction problems parametrized by a first-order theory $\mathcal{T}$.

Definition 6. Given a formula ϕ, let $\mathit{ftv}(\phi)$ be the set of free term variables and $\mathit{fpv}(\phi)$ be the set of free predicate variables in ϕ.

Definition 7. A pwCSP is defined as a pair $(\mathcal{C}, \mathcal{R})$ where $\mathcal{C}$ is a finite set of clauses of the form

$$\phi \vee \left(\bigvee_{i=1}^{\ell} X_i(\widetilde{t_i})\right) \vee \left(\bigvee_{i=\ell+1}^{m} \neg X_i(\widetilde{t_i})\right) \tag{4}$$

and $\mathcal{R} \subseteq \mathit{fpv}(\mathcal{C})$ is a set of predicate variables that are required to denote *well-founded* relations. Here, $0 \leq \ell \leq m$. Meta-variables t and ϕ range over $\mathcal{T}$-terms and $\mathcal{T}$-formulas, respectively, such that $\mathit{ftv}(\phi) = \emptyset$. Meta-variables x and X range over term and predicate variables, respectively.

A pwCSP $(\mathcal{C}, \mathcal{R})$ is called CHCs (constrained Horn clauses, [9]) if $\mathcal{R} = \emptyset$ and $\ell \leq 1$ for all clauses $c \in \mathcal{C}$. The class of CHCs has been widely studied in the verification community [12, 14, 27, 35].

Definition 8. A *predicate substitution* σ is a finite map from predicate variables X to closed predicates of the form $\lambda x_1, \ldots, x_{\mathrm{ar}(X)}.\phi$. We write $\mathrm{dom}(\sigma)$ for the domain of σ and $\sigma(\mathcal{C})$ for the application of σ to $\mathcal{C}$.

Definition 9. A predicate substitution σ is a *(genuine) solution* for $(\mathcal{C}, \mathcal{R})$ if (1) $\mathit{fpv}(\mathcal{C}) \subseteq \mathrm{dom}(\sigma)$; (2) $\models \bigwedge \sigma(\mathcal{C})$ holds; and (3) for all $X \in \mathcal{R}$, $\sigma(X)$ represents a well-founded relation, that is, $\mathrm{sort}(\sigma(X)) = (\widetilde{s}, \widetilde{s}) \to \bullet$ for some sequence $\widetilde{s}$ of sorts and there is no infinite sequence $\widetilde{v}_1, \widetilde{v}_2, \ldots$ of sequences $\widetilde{v}_i$ of values of the sorts $\widetilde{s}$ such that $\models \rho(X)(\widetilde{v}_i, \widetilde{v}_{i+1})$ for all $i \geq 1$.

Encoding termination.

Given a set of initial state $\iota(\widetilde{x})$ and a transition relation $\tau(\widetilde{x}, \widetilde{x}')$, the termination verification problem is expressed by the pwCSP $(\mathcal{C}, \mathcal{R})$ where $\mathcal{R} = \{R\}$, and $\mathcal{C}$ consists of the following clauses.

$$\iota(\widetilde{x}) \implies I(\widetilde{x}) \qquad \tau(\widetilde{x}, \widetilde{x}') \wedge I(\widetilde{x}) \implies I(\widetilde{x}')$$
$$\tau(\widetilde{x}, \widetilde{x}') \wedge I(\widetilde{x}) \implies R(\widetilde{x}, \widetilde{x}')$$

We use $\phi \implies \psi$ as syntax sugar for $\neg\phi \vee \psi$, so this is a pwCSP. The well-founded relation R asserts that τ is terminating. We also consider an invariant I for τ to avoid synthesizing ranking functions on unreachable program states.

Encoding non-termination.

We can also encode a problem of non-termination verification to pwCSP via recurrent sets [20]. For simplicity, we explain the encoding for the case of only one program variable x. We consider a recurrent set R satisfying the following conditions.

$$\iota(x) \implies R(x) \tag{5}$$
$$R(x) \implies \exists x'.\tau(x, x') \wedge R(x') \tag{6}$$

To remove $\exists$ from (6), we use the following con-

straint that is equivalent to (6).

$$R(x) \implies E(x,0) \tag{7}$$

$$E(x,x') \implies \big(\tau(x,x') \wedge R(x')\big) \vee \big(S(x',x'-1) \wedge E(x,x'-1)\big) \vee \big(S(x',x'+1) \wedge E(x,x'+1)\big) \tag{8}$$

The intuition is as follows. Given x in the recurrent set R, the relation $E(x,x')$ searches for the value of $\exists x'$ in (6). The search starts from $x'=0$ in (7), and x' is nondeterministically incremented or decremented in (8). The well-founded relation S asserts that the search finishes within finite steps. As a result, we obtain a pwCSP for non-termination defined by $(\mathcal{C},\mathcal{R})$ where $\mathcal{R}=\{S\}$ and $\mathcal{C}$ is given by (5), (7), and (the disjunctive normal form of) (8).

Example 10. Consider the following C program.

```
while(x > 0) { x = -2 * x + 9; }
```

The non-termination problem is encoded as the pwCSP $(\mathcal{C},\mathcal{R})$ where $\mathcal{R}=\{S\}$, and $\mathcal{C}$ consists of

$$x>0 \implies R(x) \qquad R(x) \implies E(x,0)$$

$$E(x,x') \implies x' = -2x+9 \wedge R(x') \vee (S(x',x'-1) \wedge E(x,x'-1)) \vee (S(x',x'+1) \wedge E(x,x'+1)).$$

The program is non-terminating when $x=3$. This is witnessed by a solution σ for $(\mathcal{C},\mathcal{R})$, which is given by $\sigma(R)(x) \coloneqq x=3$, $\sigma(E)(x,x') \coloneqq x=3 \wedge 0 \le x' \wedge x' \le 3$, and $\sigma(S)(x',x'') \coloneqq x'' = x'+1 \wedge x'' \le 3$.

4 CounterExample-Guided Inductive Synthesis (CEGIS)

We explain how CounterExample-Guided Inductive Synthesis [28] (CEGIS for short) works for a given pwCSP $(\mathcal{C},\mathcal{R})$ following [30]. Then, we add the extraction of positive/negative examples to the CEGIS architecture, which enables our decision tree-based synthesizer to use a simplified form of examples.

CEGIS proceeds through the iterative interaction between a synthesizer and a validator (Fig. 1), in which they exchange examples and candidate solutions.

Definition 11. A formula ϕ is an *example* of $\mathcal{C}$ if $\mathit{ftv}(\phi)=\emptyset$ and $\bigwedge\mathcal{C} \models \phi$ hold. Given a set $\mathcal{E}$ of examples of $\mathcal{C}$, a predicate substitution σ is a *candidate solution* for $(\mathcal{C},\mathcal{R})$ that is consistent with $\mathcal{E}$ if σ is a solution for $(\mathcal{E},\mathcal{R})$.

Synthesizer.

The input for a synthesizer is a set $\mathcal{E}$ of examples of $\mathcal{C}$ collected from previous CEGIS iterations. The synthesizer tries to find a candidate solution σ consistent with $\mathcal{E}$ instead of a genuine solution for $(\mathcal{C},\mathcal{R})$. If the candidate solution σ is found, then σ is passed to the validator. If $\mathcal{E}$ is unsatisfiable, then $\mathcal{E}$ witnesses unsatisfiability of $(\mathcal{C},\mathcal{R})$. Details of our synthesizer is described in Sect. 5.

Validator.

A validator checks whether the candidate solution σ from the synthesizer is a genuine solution of $(\mathcal{C},\mathcal{R})$ by using SMT solvers. That is, satisfiability of $\models \neg\bigwedge\sigma(\mathcal{C})$ is checked. If $\models \neg\bigwedge\sigma(\mathcal{C})$ is not satisfiable, then σ is a genuine solution of the original pwCSP $(\mathcal{C},\mathcal{R})$, so the validator accepts this. Otherwise, the validator adds new examples to the set $\mathcal{E}$ of examples. Finally, the synthesizer is invoked again with the updated set $\mathcal{E}$ of examples.

If $\models \neg\bigwedge\sigma(\mathcal{C})$ is satisfiable, new examples are constructed as follows. Using SMT solvers, the validator obtains an assignment θ to term variables such that $\models \neg\theta(\psi)$ holds for some $\psi \in \sigma(\mathcal{C})$. By (4), $\models \neg\theta(\psi)$ is a clause of the form $\models \neg\theta(\phi) \wedge \big(\bigwedge_{i=1}^{\ell} \neg\sigma(X_i)(\theta(\widetilde{t_i}))\big) \wedge \big(\bigwedge_{i=\ell+1}^{m} \sigma(X_i)(\theta(\widetilde{t_i}))\big)$. To prevent this counterexample from being found in the next CEGIS iteration again, the validator adds the following example to $\mathcal{E}$.

$$\bigvee_{i=1}^{\ell} X_i(\theta(\widetilde{t_i})) \vee \bigvee_{i=\ell+1}^{m} \neg X_i(\theta(\widetilde{t_i})) \tag{9}$$

The CEGIS architecture repeats this interaction between the synthesizer and the validator until a genuine solution for $(\mathcal{C},\mathcal{R})$ is found or $\mathcal{E}$ witnesses unsatisfiability of $(\mathcal{C},\mathcal{R})$.

Extraction of positive/negative examples.

Examples obtained in the above explanation are a bit complex to handle in our decision tree-based synthesizer: each example in $\mathcal{E}$ is a disjunction (9) of literals, which may contain multiple predicate variables.

To simplify the form of examples, we extract from $\mathcal{E}$ the sets $\mathcal{E}_X^+$ and $\mathcal{E}_X^-$ of *positive examples* (i.e., examples of the form $X(\widetilde{v})$) and *negative examples* (i.e., examples of the form $\neg X(\widetilde{v})$) for each $X \in \mathit{fpv}(\mathcal{E})$. This allows us to synthesize a predicate $\sigma(X)$ for each predicate variable $X \in \mathit{fpv}(\mathcal{E})$ separately. For simplicity, we write $\widetilde{v} \in \mathcal{E}_X^+$ and $\widetilde{v} \in \mathcal{E}_X^-$ instead of $X(\widetilde{v}) \in \mathcal{E}_X^+$ and $\neg X(\widetilde{v}) \in \mathcal{E}_X^-$.

The extraction is done as follows. We first substi-

tute for each predicate variable application $X(\widetilde{v})$ in $\mathcal{E}$ a boolean variable $b_{X(\widetilde{v})}$ to obtain a SAT problem $\mathbf{SAT}(\mathcal{E})$. Then, we use SAT solvers to obtain an assignment η that is a solution for $\mathbf{SAT}(\mathcal{E})$. If a solution η exists, then we construct positive/negative examples from η; otherwise, $\mathcal{E}$ is unsatisfiable.

Definition 12. Let η be a solution for $\mathbf{SAT}(\mathcal{E})$. For each predicate variable $X \in \mathit{fpv}(\mathcal{E})$, we define the set $\mathcal{E}_X^+$ of *positive examples* and the set $\mathcal{E}_X^+$ of *negative examples* under the assignment η by $\mathcal{E}_X^+ \coloneqq \{\widetilde{v} \mid \eta(b_{X(\widetilde{v})}) = \mathbf{true}\}$ and $\mathcal{E}_X^- \coloneqq \{\widetilde{v} \mid \eta(b_{X(\widetilde{v})}) = \mathbf{false}\}$.

Note that some of predicate variable applications $X(\widetilde{v})$ may not be assigned true nor false because they do not affect the evaluation of $\mathbf{SAT}(\mathcal{E})$. Such predicate variable applications are discarded from $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$.

Our method uses the extraction of positive and negative examples when the validator passes examples to the synthesizer. If $X \in \mathit{fpv}(\mathcal{E}) \cap \mathcal{R}$, then we apply our ranking function synthesizer to $(\mathcal{E}_X^+, \mathcal{E}_X^-)$. If $X \in \mathit{fpv}(\mathcal{E}) \setminus \mathcal{R}$, then we apply an invariant synthesizer.

We say a candidate solution σ is consistent with $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$ if $\models \sigma(X)(\widetilde{v}^+)$ and $\models \neg\sigma(X)(\widetilde{v}^-)$ hold for each predicate variable $X \in \mathit{fpv}(\mathcal{E})$, $\widetilde{v}^+ \in \mathcal{E}_X^+$, and $\widetilde{v}^- \in \mathcal{E}_X^-$. If a candidate solution σ is consistent with $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$, then σ is also consistent with $\mathcal{E}$.

Note that unsatisfiability of $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$ does not immediately implies unsatisfiability of $\mathcal{E}$ nor $(\mathcal{C}, \mathcal{R})$ because $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$ depends on the choice of the assignment η. Therefore, the CEGIS architecture need to be modified: if synthesizers find unsatisfiability of $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$, then we add the negation of an unsatisfiability core to $\mathcal{E}$ to prevent using the same assignment η again.

Note that some restricted forms of (9) have also been considered in previous work and are called implication examples in [17] and implication/negation constraints in [12]. Our extraction of positive and negative examples is applicable to the general form of (9).

5 Ranking Function Synthesis

In this section, we describe one of the main contributions, that is, our decision tree-based synthesizer, which synthesizes a candidate well-founded relation $\sigma(R)$ from a finite set $\mathcal{E}_R^+$ of examples. We assume that only positive examples are given because well-founded relations occur only positively in pwCSP for termination analysis (see Sect. 3). The aim of our synthesizer is to find a piecewise affine lexicographic ranking function $\widetilde{f}(\widetilde{x})$ for the given set $\mathcal{E}_R^+$ of examples. Below, we fix a predicate variable $R \in \mathcal{R}$ and omit the subscript $\mathcal{E}_R^+ = \mathcal{E}^+$.

5.1 Basic Definitions

To represent piecewise affine lexicographic ranking functions, we use decision trees like the one in Figure 4. Let $\widetilde{x} = (x_1, \ldots, x_n)$ be the program variables where each x_i ranges over $\mathbb{Z}$.

Definition 13. A *decision tree* D is defined by $D \coloneqq \widetilde{g}(\widetilde{x}) \mid \textbf{if } h(\widetilde{x}) \geq 0 \textbf{ then } D \textbf{ else } D$ where $\widetilde{g}(\widetilde{x}) = (g_k(\widetilde{x}), \ldots, g_0(\widetilde{x}))$ is a tuple of affine functions and $h(\widetilde{x})$ is an affine function. A *segmentation tree* S is defined as a decision tree with undefined leaves $\bot$: that is, $S \coloneqq \bot \mid \textbf{if } h(\widetilde{x}) \geq 0 \textbf{ then } S \textbf{ else } S$. For each decision tree D, we can canonically assign a segmentation tree by replacing the label of each leaf with $\bot$. This is denoted by $S(D)$. For each decision tree D, we denote the corresponding piecewise affine function by $\widetilde{f}_D(\widetilde{x}) : \mathbb{Z}^n \to \mathbb{Z}^{k+1}$.

Each leaf in a segmentation tree S corresponds to a polyhedron. We often identify the segmentation tree S with the set of leaves of S and a leaf with the polyhedron corresponding to the leaf. For example, we say something like "for each $L \in S$, $\widetilde{v} \in L$ is a point in the polyhedron L".

Suppose we are given a segmentation tree S and a set $\mathcal{E}^+$ of examples.

Definition 14. For each $L_1, L_2 \in S$, we denote the set of example transitions from L_1 to L_2 by $\mathcal{E}_{L_1,L_2}^+ \coloneqq \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+ \mid \widetilde{v} \in L_1, \widetilde{v}' \in L_2\}$. An example $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$ is *crossing* w.r.t. S if $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}_{L_1,L_2}^+$ for some $L_1 \neq L_2$, and *non-crossing* if $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}_{L,L}^+$ for some L.

Definition 15. We define the *dependency graph* $G(S, \mathcal{E}^+)$ for S and $\mathcal{E}^+$ by the graph (V, E) where vertices $V = S$ are leaves, and edges $E = \{(L_1, L_2) \mid L_1 \neq L_2, \exists(\widetilde{v}, \widetilde{v}') \in \mathcal{E}_{L_1,L_2}^+\}$ are crossing examples.

We denote the set of start points $\widetilde{v}$ and end points $\widetilde{v}'$ of examples $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$ by $\underline{\mathcal{E}^+} \coloneqq \{\widetilde{v} \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+\} \cup \{\widetilde{v}' \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+\}$.

5.2 Segmentation and (Explicit and Implicit) Cycles: One-Dimensional Case

For simplicity, we first consider the case where $\widetilde{f}(\widetilde{x}) = f(\widetilde{x}) : \mathbb{Z}^n \to \mathbb{Z}$ is a one-dimensional ranking function. Our aim is to find a ranking function $f(\widetilde{x})$ for $\mathcal{E}^+$, which satisfies $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+.\ f(\widetilde{v}) > f(\widetilde{v}')$ and $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+.\ f(\widetilde{v}) \geq 0$. If our ranking function synthesizer finds such a ranking function $f(\widetilde{x})$, then a candidate well-founded relation R_f is constructed as $R_f(\widetilde{x}, \widetilde{x}') \coloneqq f(\widetilde{x}) \geq 0 \wedge f(\widetilde{x}) > f(\widetilde{x}')$.

Our synthesizer builds a decision tree D to find a ranking function $f_D(\widetilde{x})$ for $\mathcal{E}^+$. The main question in doing so is "when and how should we refine partitions of decision trees?" To answer this question, we consider the case where there is no ranking function $f_D(\widetilde{x})$ for $\mathcal{E}^+$ with a fixed segmentation S, and classify reasons for this into three cases as follows.

Case 1: explicit cycles in examples.

We define an *explicit cycle* in $\mathcal{E}^+$ as a cycle in the graph $(\mathbb{Z}^n, \mathcal{E}^+)$. An explicit cycle witnesses that there is no ranking function for $\mathcal{E}^+$ (see e.g., Example 4).

Case 2: non-crossing examples are unsatisfiable.

The second case is when there is a leaf $L \in S$ such that no affine (not *piecewise* affine) ranking function for the set $\mathcal{E}^+_{L,L}$ of non-crossing examples exists. This prohibits the existence of piecewise affine function $f_D(\widetilde{x})$ for $\mathcal{E}^+$ with segmentation $S = S(D)$ because the restriction of $f_D(\widetilde{x})$ to $L \in S$ must be an affine ranking function for $\mathcal{E}^+_{L,L}$.

Case 3: implicit cycles in the dependency graph.

We define an *implicit cycle* by a cycle in the dependency graph $G(S, \mathcal{E}^+)$. Case 3 is the case where an implicit cycle prohibits the existence of piecewise affine ranking functions for $\mathcal{E}^+$ with the segmentation S (e.g., Example 5). If Case 1 and Case 2 do not hold but no piecewise affine ranking function for $\mathcal{E}^+$ with the segmentation S exists, then there must be an implicit cycle by (the contraposition of) the following proposition.

Proposition 16. Assume $\mathcal{E}^+$ is a set of examples that does not contain explicit cycles (i.e. Case 1 does not hold). Let S be a segmentation tree and assume that for each $L \in S$, there exists an affine ranking function $f_L(\widetilde{x})$ for $\mathcal{E}^+_{L,L}$ (i.e. Case 2 does not hold). If the dependency graph $G(S, \mathcal{E}^+)$ is acyclic, then there exists a decision tree D with the segmentation $S(D) = S$ such that $f_D(\widetilde{x})$ is a ranking function for $\mathcal{E}^+$.

Proof. By induction on the height (i.e. the length of a longest path from a vertex) of vertices in $G(S, \mathcal{E}^+)$. We construct a decision tree D as follows. If the height of $L \in S$ is 0, then we assign $f'_L(\widetilde{x}) \coloneqq f_L(\widetilde{x})$ to the leaf L where $f_L(\widetilde{x})$ is a ranking function for $\mathcal{E}^+_{L,L}$. If the height of $L \in S$ is $n > 0$, then we assign $f'_L(\widetilde{x}) \coloneqq f_L(\widetilde{x}) + c$ to the leaf L where $c \in \mathbb{Z}$ is a constant that satisfies $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{L,L'}, f_L(\widetilde{v}) + c > f'_{L'}(\widetilde{v}')$ for each cell L' with the height less than n. □

Note that the converse of Proposition 16 does not hold: the existence of implicit cycles in $G(S, \mathcal{E}^+)$ does not necessarily imply that no piecewise affine ranking function exists with the segmentation S.

5.3 Segmentation and (Explicit and Implicit) Cycles: Multi-Dimensional Lexicographic Case

We consider a more general case where $\widetilde{f}(\widetilde{x}) = (f_k(\widetilde{x}), \ldots, f_0(\widetilde{x}))$ is a multi-dimensional lexicographic ranking function and k is a fixed nonnegative integer.

Given a function $\widetilde{f}(\widetilde{x})$, we consider the well-founded relation $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ defined inductively as follows.

$$
\begin{aligned}
&R_{()}(\widetilde{x}, \widetilde{x}') \coloneqq \bot \\
&R_{(f_k, \ldots, f_0)}(\widetilde{x}, \widetilde{x}') \coloneqq \\
&\quad f_k(\widetilde{x}) \geq 0 \wedge f_k(\widetilde{x}) > f_k(\widetilde{x}') \\
&\qquad \vee f_k(\widetilde{x}) = f_k(\widetilde{x}') \wedge R_{(f_{k-1}, \ldots, f_0)}(\widetilde{x}, \widetilde{x}')
\end{aligned}
\tag{10}
$$

Our aim here is to find a lexicographic ranking function $\widetilde{f}(\widetilde{x})$ for $\mathcal{E}^+$, i.e. a function $\widetilde{f}(\widetilde{x})$ such that $R_{\widetilde{f}}(\widetilde{v}, \widetilde{v}')$ holds for each $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$. Our synthesizer does so by building a decision tree. The same argument as the one-dimensional case holds for lexicographic ranking functions.

Theorem 17 (cycle detection)**.** Assume $\mathcal{E}^+$ is a set of examples that does not contain explicit cycles. Let S be a segmentation tree and assume that for each $L \in S$, there exists an affine function $\widetilde{f}_L(\widetilde{x})$ that satisfies $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{L,L}, R_{\widetilde{f}_L}(\widetilde{v}, \widetilde{v}')$. If the dependency graph $G(S, \mathcal{E}^+)$ is acyclic, then there exists a decision tree D with the segmentation $S(D) = S$ such that $R_{\widetilde{f}_D}(\widetilde{v}, \widetilde{v}')$ holds for each $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$.

Proof. The proof is almost the same as Proposi-

Algorithm 1 Building decision trees.

Input: a set $\mathcal{E}^+$ of examples, an integer $k \geq 0$
Output: a well-founded relation R such that $\forall(\widetilde{x}, \widetilde{x}') \in \mathcal{E}^+, R(\widetilde{x}, \widetilde{x}')$

1: **if** E has a cycle **then**
2: **return** unsatisfiable
3: **end if**
4: $D \coloneqq \textsc{ResolveCase2}(E)$
5: **while** true **do**
6: $C \coloneqq \textsc{GetConstraints}(D, E)$
7: $O \coloneqq \textsc{SumAbsParams}(D)$
8: $\rho \coloneqq \textsc{Minimize}(O, C)$
9: **if** ρ is defined **then**
10: $\widetilde{f}(\widetilde{x}) \coloneqq \widetilde{f}_{\rho(D)}(\widetilde{x})$
11: **return** $R_{\widetilde{f}}$
12: **else**
13: get an unsat core in C
14: find an implicit cycle $(\widetilde{v}_1, \widetilde{v}'_1), \ldots, (\widetilde{v}_l, \widetilde{v}'_l)$ in the unsat core
15: find a cell C and two distinct points $\widetilde{v}'_i, \widetilde{v}_{i+1} \in C$ in the implicit cycle
16: add a halfspace to separate $\widetilde{v}'_i$ and $\widetilde{v}_{i+1}$ and update D
17: **end if**
18: **end while**

Algorithm 2 Resolving Case 2.

1: **function** $\textsc{ResolveCase2}(\mathcal{E}'^+)$
2: $\widetilde{f} \coloneqq \textsc{MakeAffineTemplate}(k)$
3: $C \coloneqq \textsc{GetConstraints}(\widetilde{f}, \mathcal{E}'^+)$
4: $\rho \coloneqq \textsc{GetModel}(C)$
5: **if** ρ is undefined **then**
6: $h \coloneqq \textsc{ChooseQualifier}(\mathcal{E}'^+)$
7: $D_{\geq 0} \coloneqq \textsc{ResolveCase2}(\{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^+ \mid h(\widetilde{v}) \geq 0 \wedge h(\widetilde{v}') \geq 0\})$
8: $D_{<0} \coloneqq \textsc{ResolveCase2}(\{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^+ \mid h(\widetilde{v}) < 0 \wedge h(\widetilde{v}') < 0\})$
9: **return** (**if** $h(\widetilde{x}) \geq 0$ **then** $D_{\geq 0}$ **else** $D_{<0}$)
10: **else**
11: **return** $\widetilde{f}$
12: **end if**
13: **end function**
14: **function** $\textsc{GetConstraints}(D, \mathcal{E}^+)$
15: **return** $\{R_{\widetilde{f}_D}(\widetilde{v}, \widetilde{v}') \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+\}$ where $\widetilde{f}_D$ is the tuple of piecewise affine functions corresponding to D
16: **end function**

tion 16. Here, note that if $\widetilde{f}'(\widetilde{x}) = \widetilde{f}(\widetilde{x}) + \widetilde{c}$ where $\widetilde{c}$ is a tuple of nonnegative integer constants, then $R_{\widetilde{f}'}(\widetilde{x}, \widetilde{x}')$ subsumes $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$. □

5.4 Our Decision Tree Learning Algorithm

We design a concrete algorithm based on Theorem 17. It is shown in Algorithm 1 and consists of three phases. We shall describe the three phases one by one.

Phase 1

Phase 1 (Line 1-3) detects explicit cycles in $\mathcal{E}^+$ to exclude Case 1. Here, we use a cycle detection algorithm for directed graphs.

Phase 2

Phase 2 (Line 4) detects and resolves Case 2 by using ResolveCase2 (Algorithm 2), which is a function that grows a decision tree recursively. ResolveCase2 takes non-crossing examples in a leaf, divides the leaf, and returns a *template tree* that is fine enough to avoid Case 2. Here, template trees are decision trees whose leaves are labeled by affine templates.

Algorithm 2 shows the detail of ResolveCase2. ResolveCase2 builds a template tree recursively starting from the trivial segmentation $S = \bot$ and all given examples. In each polyhedron, ResolveCase2 checks whether the set C of constraints imposed by non-crossing examples can be satisfied by an affine lexicographic ranking function on the polyhedron (Line 2-3). If the set C of constraints is not satisfiable, then ResolveCase2 chooses a halfspace $h(\widetilde{x}) \geq 0$ (Line 6) and divides the current polyhedron by the halfspace.

There is a certain amount of freedom in the choice of halfspaces. To guarantee termination of the whole algorithm, we require that the chosen halfspace h separates at least one point in $\underline{\mathcal{E}'^+} \coloneqq \{\widetilde{v} \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^+\} \cup \{\widetilde{v}' \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^+\}$ from the other points in $\underline{\mathcal{E}'^+}$. That is:

Assumption 18. If halfspace $h(\widetilde{x}) \geq 0$ is chosen in Line 6 of Algorithm 2, then there exist $\widetilde{v}, \widetilde{u} \in \underline{\mathcal{E}'^+}$

Algorithm 3 A criterion for eager qualifier selection.

1: **function** QUALITYMEASURE($h, \mathcal{E}'^{+}$)
2: $E_{++} \coloneqq \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) \geq 0 \wedge h(\widetilde{v}) \geq 0\}$
3: $E_{+-} \coloneqq \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) \geq 0 \wedge h(\widetilde{v}) < 0\}$
4: $E_{-+} \coloneqq \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) < 0 \wedge h(\widetilde{v}) \geq 0\}$
5: $E_{--} \coloneqq \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) < 0 \wedge h(\widetilde{v}) < 0\}$
6: $\widetilde{f} \coloneqq$ MAKEAFFINETEMPLATE(k)
7: $C_{+} \coloneqq$ GETCONSTRAINTS($\widetilde{f}, E_{++}$)
8: $C_{-} \coloneqq$ GETCONSTRAINTS($\widetilde{f}, E_{--}$)
9: $N_{+} \coloneqq$ MAXSMT(C_{+})
10: $N_{-} \coloneqq$ MAXSMT(C_{-})
11: **return** $N_{+} + N_{-} + (|E_{+-}| + |E_{-+}|)(1 - \mathrm{entropy}(|E_{+-}|, |E_{-+}|))$
12: **end function**

such that $h(\widetilde{v}) \geq 0$ and $h(\widetilde{u}) < 0$.

We explain two strategies (eager and lazy) to choose halfspaces that can be used to implement CHOOSEQUALIFIER. Both of them are guaranteed to terminate, and moreover, intended to yield simple decision trees.

Eager strategy.

In the eager strategy, we eagerly generate a finite set H of halfspaces from the set $\mathcal{E}^{+}$ of all examples beforehand and choose the best one from H with respect to a certain quality measure. To satisfy Assumption 18, H are generated so that any two points $\widetilde{u}, \widetilde{v} \in \underline{\mathcal{E}^{+}}$ can be separated by some halfspace $(h(\widetilde{x}) \geq 0) \in H$.

For example, we can use intervals $H = \{\pm(x_i - a_i) \geq 0 \mid i = 1, \ldots, n \wedge (a_1, \ldots, a_n) \in \underline{\mathcal{E}^{+}}\}$ and octagons $H = \{\pm(x_i - a_i) \pm (x_j - a_j) \geq 0 \mid i \neq j \wedge (a_1, \ldots, a_n) \in \underline{\mathcal{E}^{+}}\}$ where $\widetilde{x} = (x_1, \ldots, x_n)$. For any input $\mathcal{E}'^{+} \subseteq \mathcal{E}^{+}$ of RESOLVECASE2, intervals and octagons satisfy $\emptyset \neq H' \coloneqq \{h(\widetilde{x}) \geq 0 \mid \exists \widetilde{v}, \widetilde{u} \in \mathcal{E}'^{+}. h(\widetilde{v}) \geq 0 \wedge h(\widetilde{u}) < 0\}$, so Assumption 18 is satisfied by choosing the best halfspace with respect to the quality measure from H'.

For each halfspace $(h(\widetilde{x}) \geq 0) \in H'$, we calculate QUALITYMEASURE in Algorithm 3, and choose one that maximizes QUALITYMEASURE($h, \mathcal{E}'^{+}$). QUALITYMEASURE($h, \mathcal{E}'^{+}$) calculates the sum of the maximum number of satisfiable constraints in each leaf divided by $h(\widetilde{x}) \geq 0$ plus an additional term $(|E_{+-}| + |E_{-+}|)(1 - \mathrm{entropy}(|E_{+-}|, |E_{-+}|))$ where $\mathrm{entropy}(x, y) = -\frac{x}{x+y} \log_2 \frac{x}{x+y} - \frac{y}{x+y} \log_2 \frac{y}{x+y}$. Therefore, the term $(|E_{+-}| + |E_{-+}|)(1 - \mathrm{entropy}(|E_{+-}|, |E_{-+}|))$ is close to $|E_{+-}| + |E_{-+}|$ if almost all examples in $E_{+-} \cup E_{-+}$ cross h in the same direction and close to 0 if $|E_{+-}|$ is almost equal to $|E_{-+}|$.

Lazy strategy.

In the lazy strategy, we lazily generate halfspaces. We divide the current polyhedron so that non-crossing examples in the cell point to almost the same direction.

First, we label states that occur in $\mathcal{E}^{+}_{C,C}$ as follows. We find a direction that most examples in C point to by solving the MAX-SMT $\vec{a} \coloneqq \max_{\vec{a}} \left|\{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^{+}_{C,C} \mid \vec{a} \cdot (\widetilde{v} - \widetilde{v}') > 0\}\right|$. For each $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^{+}_{C,C}$, we label two points $\widetilde{v}, \widetilde{v}'$ with $+1$ if $\vec{a} \cdot (\widetilde{v} - \widetilde{v}') > 0$ and with -1 otherwise.

Then we apply weighted C-SVM to generate a hyperplane that separates most of the positive and negative points. To guarantee termination of Algorithm 1, we avoid "useless" hyperplanes that classify all the points by the same label. If we obtain such a useless hyperplane, then we undersample a majority class and apply C-SVM again. By undersampling suitably, we eventually get linearly separable data with at least one positive point and one negative point.

Note that since coefficients of hyperplanes extracted from C-SVM are floating point numbers, we have to approximate them by hyperplanes with rational coefficients. This is done by truncating continued fraction expansions of coefficients by a suitable length.

Phase 3

In Line 5-18 of Algorithm 1, we further refine the segmentation $S(D)$ to resolve Case 3. Once Case 2 is resolved by RESOLVECASE2, Case 2 never holds even after refining $S(D)$ further. This enables to separate Phases 2 and 3.

Given a template tree D, we consider the set C of constraints on parameters in D that claims $\widetilde{f}_D(\widetilde{x})$ is a ranking function for $\mathcal{E}^{+}$ (Line 6).

If C is satisfiable, we use an SMT solver to obtain a solution of C (i.e. an assignment ρ of integers to parameters) while minimizing the sum of absolute values of unknown parameters in D at the same time (Line 8). This minimization is intended to give a simple candidate ranking function. The so-

lution ρ is used to instantiate the template tree D (Line 11).

If C cannot be satisfied, there must be an implicit cycle in the dependency graph $G(S(D), \mathcal{E}^+)$ by Theorem 17. The implicit cycle can be found in an unsatisfiable core of C. We refine the segmentation of D to cut the implicit cycle in Line 16. To guarantee termination, we choose a halfspace satisfying the following assumption, which is similar to Assumption 18.

Assumption 19. If halfspace $h(\widetilde{x}) \geq 0$ is chosen in Line 16 of Algorithm 1, then there exist $\widetilde{v}, \widetilde{u} \in \underline{\mathcal{E}^+}$ such that $h(\widetilde{v}) \geq 0$ and $h(\widetilde{u}) < 0$.

We have two strategy (eager and lazy) to refine the segmentation of D.

In eager strategy, we choose a halfspace $(h(\widetilde{x}) \geq 0) \in H$ that separates two distinct points $\widetilde{v}'_i$ and $\widetilde{v}_{i+1}$ in the implicit cycle. In doing so, we want to reduce the number of implicit cycles in $G(S(D), \mathcal{E}^+)$, but adding a new halfspace may introduce new implicit cycles if there exists $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{C,C}$ that crosses the new border from the side of $\widetilde{v}'_i$ to the side of $\widetilde{v}_{i+1}$. Therefore, we choose a hyperplane that minimizes the number of new crossing examples.

In lazy strategy, we use an SMT solver to find a hyperplane $h(\widetilde{x}) \in H$ that separates $\widetilde{v}'_i$ and $\widetilde{v}_{i+1}$ and minimizes the number of new crossing examples.

Termination

Assumption 18 and Assumption 19 guarantees that every leaf in $S(D)$ contains at least one point in the finite set $\underline{\mathcal{E}^+}$. Because the number of leaves in $S(D)$ strictly increases after each iteration of Phase 2 and Phase 3, we eventually get a segmentation $S(D)$ where each $L \in S(D)$ contains only one point in $\underline{\mathcal{E}^+}$ in the worst case. Since we have excluded Case 1 at the beginning, Theorem 17 guarantees the existence of ranking function with the segmentation $S(D)$. Therefore, the algorithm terminates within $|\underline{\mathcal{E}^+}|$ times of refinement.

Theorem 20. If Assumption 18 and Assumption 19 hold, then Algorithm 1 terminates. If Algorithm 1 returns a piecewise affine lexicographic function $\widetilde{f}(\widetilde{x})$, then the function satisfies $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ for each $(\widetilde{x}, \widetilde{x}') \in \mathcal{E}^+$ where $\mathcal{E}^+$ is the input of the algorithm.

5.5 Improvement by Degenerating Negative Values

There is another way to define well-founded relation from the tuple $\widetilde{f}(\widetilde{x}) = (f_k(\widetilde{x}), \ldots, f_0(\widetilde{x}))$ of functions, that is, the well-founded relation $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ defined inductively by $R'_{()}(\widetilde{x}, \widetilde{x}') \coloneqq \bot$ and $R'_{(f_k,\ldots,f_0)}(\widetilde{x}, \widetilde{x}') \coloneqq f_k(\widetilde{x}) \geq 0 \wedge f_k(\widetilde{x}) > f_k(\widetilde{x}') \vee \big(f_k(\widetilde{x}') < 0 \vee f_k(\widetilde{x}) = f_k(\widetilde{x}')\big) \wedge R'_{(f_{k-1},\ldots,f_0)}(\widetilde{x}, \widetilde{x}')$.

In this definition, we loosen the equality $f_i(\widetilde{x}) = f_i(\widetilde{x}')$ (where $i = 1, \ldots, k$) of the usual lexicographic ordering (10) to $f_i(\widetilde{x}') < 0 \vee f_i(\widetilde{x}) = f_i(\widetilde{x}')$. This means that once $f_i(\widetilde{x})$ becomes negative, $f_i(\widetilde{x})$ must stay negative but the value do not have to be the same, which is useful for the synthesizer to avoid complex candidate lexicographic ranking functions and thus improves the performance.

However, if we use this well-founded relation $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ instead of $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ in (10), then Theorem 17 fails because $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ is not necessarily subsumed by $R'_{\widetilde{f}+\widetilde{c}}$ where $\widetilde{c} = (c_k, \ldots, c_0)$ is a non-negative constant (see the proof of Proposition 16 and Theorem 17). As a result, there is a chance that no implicit cycle can be found in line 14 of Algorithm 1. Therefore, when we use $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$, we modify Algorithm 1 so that if no implicit cycle can be found in line 14, then we fall back on the former definition of $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ and restart Algorithm 1.

6 Implementation and Evaluation

Implementation.

We implemented a constraint solver MuVal that supports invariant synthesis and ranking function synthesis. For invariant synthesis, we apply an ordinary decision tree learning (see [12, 14, 18, 22, 35] for existing techniques). For ranking function synthesis, we implemented the algorithm in Sect. 5 with both eager and lazy strategies for halfspace selection. Our synthesizer uses well-founded relation explained in Sect. 5.5. Given a benchmark, we run our solver for both termination and non-termination verification in parallel, and when one of the two returns an answer, we stop the other and use the answer. MuVal is written in OCaml and uses Z3 as an SMT solver backend. We used clang and llvm2kittel [1] to convert C benchmarks to T2 [3] format files, which are then translated to pwCSP by MuVal.

Table 2: Numbers of solved benchmarks

	Yes	No	TO	U
MuVal (eager)	204	89	42	0
MuVal (lazy)	200	84	51	0
AProVE	216	100	16	3
iRankFinder	208	92[†1]	0	34
Ultimate Automizer	180	83	2	70

Experiments.

We evaluated our implementation MuVal on C benchmarks from Termination Competition 2020 (C Integer) [4]. We compared our tool with AProVE [10, 13], iRankFinder [7], and Ultimate Automizer [21]. Experiments are conducted on StarExec [2] (CentOS 7.7 (1908) on Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz (2393 MHZ) with 263932744 kB main memory). The time limit was 300 seconds.

Results.

Results are shown in Table 2. Yes/No/TO/U means the number of benchmarks that these tools could verify termination/could verify non-termination/could not answer within 300 seconds and timed out (TimeOut)/gave up before 300 seconds (Unknown), respectively. We also show scatter plots of runtime in Fig. 7.

MuVal was able to solve more benchmarks than Ultimate Automizer. Compared to iRankFinder, MuVal solved slightly fewer benchmarks, but was faster in a large number of benchmarks: 265 benchmarks were solved faster by MuVal, 68 by iRankFinder, and 2 were not solved by both tools within 300 seconds (here, we regard U (unknown) as 300 seconds). Compared to AProVE, MuVal solved fewer benchmarks. However, there are several benchmarks that MuVal could solve but AProVE could not. Among them is "TelAviv-Amir-Minimum_true-termination.c", which does require piecewise affine ranking functions. MuVal found a ranking function $f(x, y) = \textbf{if } x - y \geq 0 \textbf{ then } y \textbf{ else } x$, while AProVE timed out.

We also observed that using CEGIS with transition examples itself showed its strengths even for benchmarks that do not require piecewise affine ranking functions. Notably, there are three benchmarks that MuVal could solve but the other tools could not; they are examples that do not require segmentations. Further analysis of these benchmarks indicates the following strengths of our framework: (1) the ability to handle nonlinear constraints (to some extent) thanks to the example-based synthesis and the recent development of SMT solvers; and (2) the ability to find a long lasso-shaped non-terminating trace assembled from multiple transition examples. See Appendix A for details.

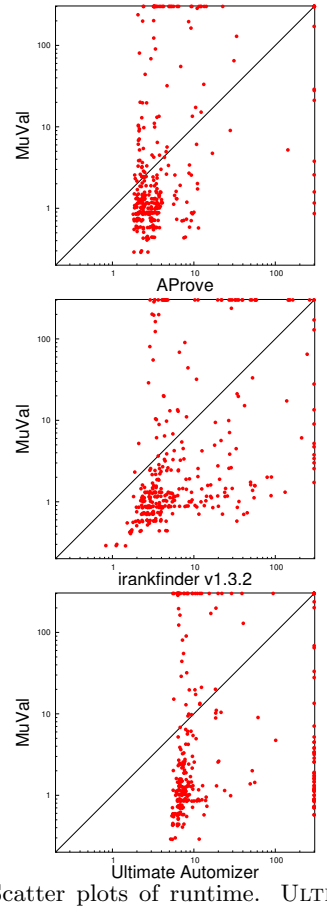


Fig. 7: Scatter plots of runtime. Ultimate Automizer and AProVE sometimes gave up before the time limit, and such cases are regarded as 300s.

7 Related Work

There are a bunch of works that synthesize ranking functions via constraint solving. Among them is a counterexample-guided method like CEGIS [28]. CEGIS is sound but not guaranteed to be complete in general: even if a given constraint has a

†1 We removed one benchmarks from the result of iRankFinder because the answer was wrong.

solution, CEGIS may fail to find the solution. A complete method for ranking function synthesis is proposed in [19]. They collect only extremal counterexamples instead of arbitrary transition examples to avoid infinitely many examples. A limitation of their method is that the search space is limited to (lexicographic) affine ranking functions.

Another counterexample-guided method is proposed in [32] and implemented in SeaHorn. This method can synthesize piecewise affine functions, but their approach is quite different from ours. Given a program, they construct a *safety* property that the number of loop iterations does not exceed the value of a candidate ranking function. The safety property is checked by a verifier. If it is violated, then a trace is obtained as a counterexample and the candidate ranking function is updated by the counterexample. The main difference from our method is that their method uses trace examples while our method uses transition examples (which is less expensive to handle). FreqTerm [15] also uses the connection to safety property, but they exploit syntax-guided synthesis for synthesizing ranking functions.

Aside from counterexample-guided methods, constraint solving is widely studied for affine ranking functions [26], lexicographic affine ranking functions [5, 7, 23], and multiphase affine ranking functions [6, 8]. Their implementation includes RankFinder and iRankFinder. Farkas' lemma or Motzkin's transposition theorem are often used as a tool to transform $\exists\forall$-constraints to $\exists$-constraints. However, when we apply this technique to piecewise affine ranking functions, we get nonlinear constraints [23].

Abstract interpretation is also applied to segmented synthesis of ranking functions and implemented in FuncTion [31, 33, 34]. In this series of work, decision tree representation of ranking functions is used in [34] for better handling of disjunctions. Compared to their work, we believe that our method is more easily extensible to other theories than linear integer arithmetic as long as the theories are supported by SMT solvers (although such extensions are out of the scope of this paper).

Other state-of-the-art termination verifiers include the following. Ultimate Automizer [21] is an automata-based method. It repeatedly finds a trace and computes a termination argument that contains the trace until termination arguments cover the set of all traces. Büchi automata are used to handle such traces. AProVE [10,13] is based on term rewriting systems.

8 Conclusions and Future Work

In this paper, we proposed a novel decision tree-based synthesizer for ranking functions, which is integrated into the CEGIS architecture. The key observation here was that we need to cope with explicit and implicit cycles contained in given examples. We designed a decision tree learning algorithm using the theoretical observation of the cycle detection theorem. We implemented the framework and observed that its performance is comparable to state-of-the-art termination analyzers. In particular, it solved three benchmarks that no other tool solved, a result that demonstrates the potential of the current combination of CEGIS, segmented synthesis, and transition examples.

We plan to extend our ranking function synthesizer to a synthesizer of piecewise affine ranking supermartingales. Ranking supermartingales [11] are probabilistic version of ranking functions and used for verification of almost-sure termination of probabilistic programs.

We also plan to implement a mechanism to automatically select a suitable set of halfspaces with which decision trees are built. In our ranking function synthesizer, intervals/octagons/octahedron/polyhedra can be used as the set of halfspaces. However, selecting an overly expressive set of halfspaces may cause the problem of overfitting [24] and result in poor performance. Therefore, applying heuristics that adjusts the expressiveness of halfspaces based on the current examples may improve the performance of our tool.

Acknowledgement.

We thank Andrea Peruffo and the anonymous referees for many suggestions. This work was supported by JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603) and JSPS KAKENHI Grant Numbers 20H04162, 20H05703, 19H04084, and 17H01720.

References

[1] : llvm2KITTeL, https://github.com/hkhlaaf/llvm2kittel.

[2] : StarExec, https://www.starexec.org.
[3] : T2 TEMPORAL LOGIC PROVER, https://github.com/mmjb/T2.
[4] : Termination Competition 2020: C Integer, https://termcomp.github.io/Y2020/job_41519.html.
[5] Alias, C., Darte, A., Feautrier, P., and Gonnord, L.: Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs, *SAS '10*, Springer, 2010, pp. 117–133.
[6] Ben-Amram, A. M., Doménech, J. J., and Genaim, S.: Multiphase-Linear Ranking Functions and Their Relation to Recurrent Sets, *SAS '19*, Springer, 2019, pp. 459–480.
[7] Ben-Amram, A. M. and Genaim, S.: Ranking Functions for Linear-Constraint Loops, *Journal of the ACM*, Vol. 61, No. 4(2014).
[8] Ben-Amram, A. M. and Genaim, S.: On Multiphase-Linear Ranking Functions, *CAV '17*, Springer, 2017, pp. 601–620.
[9] Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A.: Horn Clause Solvers for Program Verification, *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, LNCS, Vol. 9300, Springer, 2015, pp. 24–51.
[10] Brockschmidt, M., Ströder, T., Otto, C., and Giesl, J.: Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode, *FoVeOOS '11*, LNCS, Vol. 7421, Springer, 2012, pp. 123–141.
[11] Chakarov, A. and Sankaranarayanan, S.: Probabilistic Program Analysis with Martingales, *Computer Aided Verification*, Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Sharygina, N., and Veith, H.(eds.), Vol. 8044, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 511–526.
[12] Champion, A., Chiba, T., Kobayashi, N., and Sato, R.: ICE-Based Refinement Type Discovery for Higher-Order Functional Programs, *TACAS '18*, LNCS, Vol. 10805, Springer, 2018, pp. 365–384.
[13] Emmes, F., Enger, T., and Giesl, J.: Proving Non-looping Non-termination Automatically, *IJCAR '12*, LNCS, Vol. 7364, Springer, 2012, pp. 225–240.
[14] Ezudheen, P., Neider, D., D'Souza, D., Garg, P., and Madhusudan, P.: Horn-ICE Learning for Synthesizing Invariants and Contracts, *Proceedings of the ACM on Programming Languages*, Vol. 2, No. OOPSLA(2018), pp. 131:1–131:25.
[15] Fedyukovich, G., Zhang, Y., and Gupta, A.: Syntax-Guided Termination Analysis, *CAV '18*, LNCS, Vol. 10981, Springer, 2018, pp. 124–143.
[16] Floyd, R. W.: Assigning Meanings to Programs, *Proceedings of Symposium on Applied Mathematics*, Vol. 19(1967), pp. 19–32.
[17] Garg, P., Löding, C., Madhusudan, P., and Neider, D.: ICE: A Robust Framework for Learning Invariants, *CAV '14*, Springer, 2014, pp. 69–87.
[18] Garg, P., Neider, D., Madhusudan, P., and Roth, D.: Learning Invariants Using Decision Trees and Implication Counterexamples, *POPL '16*, ACM, 2016, pp. 499–512.
[19] Gonnord, L., Monniaux, D., and Radanne, G.: Synthesis of Ranking Functions Using Extremal Counterexamples, *PLDI '15*, ACM, 2015, pp. 608–618.
[20] Gupta, A., Henzinger, T. A., Majumdar, R., Rybalchenko, A., and Xu, R.-G.: Proving non-termination, *POPL '08*, ACM, 2008, pp. 147–158.
[21] Heizmann, M., Hoenicke, J., and Podelski, A.: Termination Analysis by Learning Terminating Programs, *CAV '14*, Springer, 2014, pp. 797–813.
[22] Krishna, S., Puhrsch, C., and Wies, T.: Learning Invariants using Decision Trees, *CoRR*, Vol. abs/1501.04725(2015).
[23] Leike, J. and Heizmann, M.: Ranking Templates for Linear Loops, *TACAS '14*, LNCS, Vol. 8413, Springer, 2014, pp. 172–186.
[24] Padhi, S., Millstein, T. D., Nori, A. V., and Sharma, R.: Overfitting in Synthesis: Theory and Practice, *CAV '19*, LNCS, Vol. 11561, Springer, 2019, pp. 315–334.
[25] Padhi, S., Sharma, R., and Millstein, T. D.: Data-Driven Precondition Inference with Learned Features, *PLDI '16*, 2016, pp. 42–56.
[26] Podelski, A. and Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions, *VMCAI '04*, LNCS, Vol. 2937, Springer, 2004, pp. 239–251.
[27] Satake, Y., Unno, H., and Yanagi, H.: Probabilistic Inference for Predicate Constraint Satisfaction, *Proceedings of AAAI 2020*, 2020.
[28] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V.: Combinatorial Sketching for Finite Programs, *ASPLOS XII*, ACM, 2006, pp. 404–415.
[29] Unno, H., Satake, Y., Terauchi, T., and Koskinen, E.: Program Verification via Predicate Constraint Satisfiability Modulo Theories, *CoRR*, Vol. abs/2007.03656(2020).
[30] Unno, H., Terauchi, T., and Koskinen, E.: Constraint-based Relational Verification, *CAV '21*, Springer, 2021.
[31] Urban, C.: The Abstract Domain of Segmented Ranking Functions, *SAS '13*, LNCS, Vol. 7935, Springer, 2013, pp. 43–62.
[32] Urban, C., Gurfinkel, A., and Kahsai, T.: Synthesizing Ranking Functions from Bits and Pieces, *TACAS '16*, Springer, 2016, pp. 54–70.
[33] Urban, C. and Miné, A.: An Abstract Domain to Infer Ordinal-Valued Ranking Functions, *ESOP*

'14, Springer, 2014, pp. 412–431.
[34] Urban, C. and Miné, A.: A Decision Tree Abstract Domain for Proving Conditional Termination, *SAS '14*, Springer, 2014, pp. 302–318.
[35] Zhu, H., Magill, S., and Jagannathan, S.: A Data-driven CHC Solver, *PLDI '18*, ACM, 2018, pp. 707–721.

A Detailed Discussion of The Experiment

We show some of the benchmarks from Termination Competition 2020 that are solved by MuVal but not by AProVE, Ultimate Automizer, and iRankFinder. We also discuss the strength of our method.

A benchmark containing a nonlinear operation.

Fig. 8a shows one of the benchmarks that contains a nonlinear operation `y = y * y` but admits a linear ranking function. Although handling nonlinear operation is difficult in general, MuVal was able to verify termination. The reason can be understood as follows. (1) Our validator can find transition examples for this benchmark thanks to the recent development of SMT solvers. (2) Our synthesizer can work as usual because it is example-based.

A benchmark that requires a long lasso-shaped non-terminating trace to prove non-termination.

Fig. 8b is a benchmark that is non-terminating. To prove non-termination of this benchmark, we need to find a long lasso-shaped trace that ends in the self loop when `i` = `range` = 0. Finding such a long lasso is difficult in general, but MuVal was able to find one in this benchmark: during CEGIS iterations, MuVal found the self loop (i.e. explicit cycle) and then collected transition examples that were needed to prove reachability of cycles.

Our method can naturally collect transition examples "backward" from the self loop (an explicit cycle). At the same time, our method collects transition examples "forward" from an initial state (in the benchmark of Fig. 8b, the pair of `i` = 10 and `range` = 20 is an initial state of the while loop). When transition examples that are collected backward and forward form a trace to the self loop, our method notice that the benchmark is non-terminating.

```
int main() {
  int x = ?;
  int y = 2;
  int res = 1;
  if (x < 0 || y < 1) { }
  else {
    while (x > y) {
      y = y*y;
      res = 2*res;
    }
  }
}
```

(a) A benchmark containing a nonlinear operation.

```
int main() {
  int i = ?;
  int range = 20;
  while (0 <= i && i <= range) {
    if (!(0 == i && i == range)) {
      if (i == range) {
        i = 0;
        range = range-1;
      } else {
        i = i+1;
      }
    }
  }
}
```

(b) A benchmark that requires a long lasso to prove non-termination

Fig. 8: Some of the benchmarks