Journal of Computer Languages 75 (2023) 101206

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Quantitative program sketching using decision tree-based lifted analysis

Aleksandar S. Dimovski

Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia

ARTICLE INFO

Keywords:
Quantitative program sketching
Lifted static analysis
Abstract interpretation
Software product lines
Decision tree lifted domain

ABSTRACT

We present a novel approach for resolving numerical program sketches under Boolean and quantitative objectives. The input is a program sketch, which represents a partial program with missing numerical parameters (holes). The aim is to automatically synthesize values for the parameters, such that the resulting complete program satisfies: a *Boolean (qualitative) specification* given in the form of assertions; and a *quantitative specification* that estimates the number of execution steps to termination and which the synthesizer is expected to optimize.

To address the above quantitative sketching problem, we encode a program sketch as a program family (a.k.a. Software Product Line) and use the specifically designed lifted analysis algorithms based on abstract interpretation for efficiently analyzing program families with numerical features. The elements of the lifted analysis domain are *decision trees*, in which decision nodes are labeled with linear constraints defined over numerical features and leaf nodes belong to an existing single-program analysis domain. First, we transform a program sketch into a program family, such that numerical holes correspond to numerical features and all possible sketch realizations correspond to variants in the program family. Then, we use a combination of forward (numerical) and backward (quantitative termination) lifted analysis of program families to find the variants (family members) that satisfy all assertions, and moreover are optimal with respect to the given quantitative objective. Such obtained variants represent the "correct & optimal" realizations of the given program sketch.

We present a prototype implementation of our approach within the FamilySketcher tool for resolving C sketches with numerical data types. We have evaluated our approach on a set of numerical benchmarks, and experimental results confirm the effectiveness of our approach. In some cases, our approach provides speedups against the well-known sketching tool Sketch and resolves some numerical benchmarks that Sketch cannot handle.

1. Introduction

Sketching [1,2] is one of the earliest and successful forms of program synthesis [3]. A *sketch* is a partial program with missing numerical expressions called *holes* to be discovered by the synthesizer. The inputs to the program sketching problem are a sketch and a specification of the required complete program in the form of assertions. Previous approaches for solving the program sketching problem [1,2,4] automatically synthesize integer constant values for the holes so that the resulting complete program satisfies *Boolean (qualitative) properties* in the form of assertions. However, the need for considering combined Boolean and quantitative properties is prominent in many applications. Still, quantitative properties have been largely missing from previous approaches for program sketching. In particular, there has been no possibility for measuring the "goodness" of solutions. Boolean properties are used to define minimal requirements for the synthesized complete programs, that is, they should at least satisfy the given assertions. Still, there are usually many different complete programs that satisfy the Boolean properties, and some of them may be preferred over the others. Therefore, it is important to define synthesis algorithms, which construct complete programs (solutions) that not only meet the Boolean properties, but are also optimal with respect to a given quantitative objective [5,6]. This is so-called *quantitative sketching problem*.

In this paper, we use *lifted static analysis* based on abstract interpretation for program families (a.k.a. Software Product Lines) [7] to solve this quantitative sketching problem. *Abstract interpretation* [8,9] is a general theory for approximating the semantics of programs. It represents a powerful technique for deriving approximate, albeit computable static analyses, by using fully automatic algorithms. The *key observation* is that all possible sketch realizations constitute a program family, where each numerical hole is represented as a numerical feature. A *program family* describes a set of similar programs as *variants* of some common code base [10,11]. At compile-time, a variant of

E-mail address: aleksandar.dimovski@unt.edu.mk.

https://doi.org/10.1016/j.cola.2023.101206
Received 14 September 2022; Received in revised form 30 March 2023; Accepted 31 March 2023
Available online 6 April 2023

a program family is derived by assigning concrete values to a set of *features* (configuration options) relevant for it, and only then is this variant compiled or interpreted. Program families (often in C) enriched with compile-time configurability by the C preprocessor CPP [7,12] are today widely used in open-source projects and industry [12]. By using the proposed transformation from program sketches to program families, we translate the quantitative sketching problem to selecting those variants (family members) from the corresponding program family that satisfy all assertions and are optimal with respect to the given quantitative objective. As a *quantitative objective* we consider here the sufficient preconditions inferred by a *quantitative termination analysis* that estimates the efficiency of a program by counting upper-bounds on the number of execution steps to termination. More specifically, we use a combination of forward and backward lifted analysis to solve this problem. The forward numerical lifted analysis infers numerical invariants for all members of a program family, thus finding the "correct" variants (family members) that satisfy all assertions. Subsequently, the backward termination lifted analysis is performed on a sub-family of "correct" variants to infer piecewise-defined ranking functions, which provide upper-bounds on the number of execution steps to termination. The "correct" variants with minimal ranking function are reported as optimal complete programs that solve the original quantitative sketching problem.

However, the automated analysis of program families for finding a "correct & optimal" variant is challenging since the family size (i.e., number of variants) typically grows exponentially in the number of features. This is particularly apparent in the case of program families that contain numerical features with big domains, thus admitting astronomic family sizes. Program sketching is also affected by this problem, since the family size corresponds to the space of sketch realizations. A naive enumerative (brute-force) approach, which analyzes each individual variant of the program family by an existing single-program static analyzer based on abstract interpretation, has been shown to be very inefficient for larger families [13,14]. This brute-force approach has to preprocess, build control flow graph, and execute the fixed point iterative algorithm once for each possible variant.

To overcome this efficiency problem of the brute-force approach, we use the specifically designed lifted static analysis algorithms [14–21] to speed up the process of finding the required (correct & optimal) variants. They analyze the common code base of a program family directly, without preprocessing any of variants explicitly. Thus, the lifted analysis builds the control flow graph and executes the fixed point iterative algorithm only once per family. In particular, we use an abstract interpretation-based lifted analysis of program families with numerical features [18], where *sharing* is explicitly possible between equivalent analysis elements corresponding to different variants. In effect, the lifted analysis and the brute-force enumeration approach produce identical (precision-wise) analysis results, but the lifted analysis is more efficient thanks to the improved representation of the underlying analysis elements. This is achieved by using a specialized *decision tree lifted domain* [18] that provides a symbolic and compact representation of the lifted analysis elements. More precisely, the elements of the lifted domain are *decision trees*, in which decision nodes are labeled with linear constraints over features, while leaf nodes belong to an existing single-program analysis domain (e.g., some numerical domain [9] or the termination domain [22,23]). The decision trees recursively partition the space of variants (i.e., the space of possible combinations of feature's values), whereas the program properties at the leaves provide analysis information corresponding to each partition (i.e., to those variants that satisfy the constraints along the path to the given leaf node), thus producing disjunctive analysis results corresponding to all partitions. This way, the forward (numerical) lifted analysis partitions the given family into: "correct", "incorrect", and "I don't know" (inconclusive) sub-families (sets of variants) with respect to the given assertions. Phrased in terms of program sketching, the analysis partitions the family into a sub-family with correct sketch completions; a sub-family with incorrect sketch completions; and a sub-family with inconclusive sketch completions for which indefinite answer is obtained due to the precision loss (over-approximation) introduced in the static analysis by abstraction. The backward (quantitative termination) lifted analysis additionally partitions the "correct" sub-family with respect to the estimated number of execution steps to termination. Because of its special structure and possibilities for sharing of equivalent analysis results, the decision tree-based lifted analyses are able to converge to a solution very fast even for program families (sketches) that contain numerical features (holes) with large domains, thus giving rise to astronomical search spaces. This is particularly true for sketches in which holes appear in linear expressions that can be exactly represented in the underlying numerical domains used in the decision trees (e.g., intervals, octagons, polyhedra). In those cases, we can extend the standard lifted analysis for classical program families and design more efficient lifted analysis algorithms with improved transfer functions for analyzing assignments and tests in which holes in linear expressions are present. Note that transfer functions in abstract analyses [9,14] capture the effect of abstractly analyzing statements in a given abstract element.

Moreover, we also examine various quantitative cost-sensitive analyses and the corresponding quantitative objectives induced by assigning different costs on various operations of the language. This way, we can define an endless variety of quantitative cost-sensitive analyses parameterized by the resource one needs to observe and how different operations of the language consume that resource. In this work, we also show how to solve the quantitative sketching problem with respect to a specific quantitative objective, which is defined by a cost model showing the overhead of executing different operations of the language.

We have implemented our approach in a prototype program synthesizer, called FamilySketcher [4]. The numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library [24] are used as parameters of the underlying decision trees. FamilySketcher calls the Z3 SMT solver [25] to solve the optimization problem that represents the given quantitative objective. We illustrate this approach for automatic completion of various numerical C sketches with numerical data types from the Sketch project [1,2], SV-COMP (https://sv-comp.sosy-lab.org/), and from the Syntax-Guided Synthesis Competition (https://sygus.org/). We also compare performances of our approach against the most popular sketching tool Sketch [1,2] and the brute-force enumeration approach that checks for correctness and optimality all sketch realizations one by one.

In summary, this work makes the following contributions:

1. We combine forward and backward lifted analyses to resolve numerical program sketches with respect to both Boolean and quantitative specifications;
2. We consider various quantitative cost-sensitive specifications induced by a cost model that specifies the overhead of executing different operations of the language;
3. We implement our approach in the FamilySketcher tool, which uses numerical domains from the APRON library as parameters and the Z3 tool for solving the underlying (linear) optimization problem;
4. We evaluate our approach and compare its performances with the popular Sketch tool and the brute-force enumeration approach.

This work extends and revises the conference article [26]. We make the following extensions here: (1) we introduce novel quantitative cost-sensitive objectives and solve the quantitative sketching problem with respect to them; (2) we expand the evaluation by considering more benchmarks and by extending the performance results; (3) we provide formal proofs for all main results in the work; (4) we provide additional illustrations, explanations, and examples. The paper proceeds with motivating examples that illustrate our new approaches for quantitative sketching. The languages for writing sketches and program families,

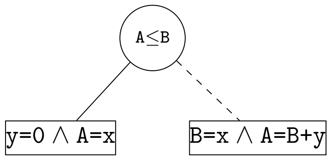


Fig. 1. Lifted numerical invariant at location ⑤ of Loop1A (solid edges = true, dashed edges = false).

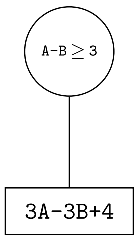


Fig. 2. Lifted ranking function at location ① of Loop1A.

as well as rules for transforming sketches into program families are introduced in Section 3. Section 4 defines the algorithms for decision tree-based lifted analysis, while Section 5 introduces a variety of quantitative cost-sensitive analyses. The synthesis algorithm for solving the quantitative sketching problem is described in Section 6. Section 7 presents the evaluation on benchmarks taken from SV-COMP, SyGuS, and Sketch. Finally, we discuss related work and conclude.

2. Motivating examples

Let us consider the following Loop1A sketch taken from Syntax-Guided Synthesis Competition (https://sygus.org/) [3]:

```
void  main() {
①     int x := ??₁, y := 0;
②     while ⓗ(x > ??₂) {
③       x := x-1;
④       y := y+1; }
⑤     assert(y > 2); //assert(y < 8); }
```

It contains two numerical holes, denoted by $??_1$ and $??_2$. The synthesizer should replace the holes with constants from $\mathbb{Z}$, such that the synthesized program satisfies the assertion at location ⑤ under all possible inputs. Moreover, we want to select the most time-efficient correct program, i.e. the one that terminates in the minimum number of execution steps.

We transform the Loop1A sketch to a program family, which contains two numerical features A and B with domains $[\mathtt{Min}, \mathtt{Max}] \subseteq \mathbb{Z}$.[1] Since both holes in the Loop1A sketch occur in (linear) expressions that can be exactly represented in numerical domains (e.g. intervals), the Loop1A program family is obtained by replacing the two holes $??_1$ and $??_2$ with the features A and B (see Fig. 7). The total number of variants that can be generated from this family is $(\mathtt{Max} - \mathtt{Min} + 1)^2$, so that each variant corresponds to one possible sketch realization. For example, the variant in which $(\mathtt{A} = 0 \wedge \mathtt{B} = 0)$ corresponds to the sketch realization in which $??_1$ and $??_2$ are both set to 0. We perform a *forward numerical lifted analysis* based on decision trees [18] of the Loop1A program family. The input decision tree at location ① has only one leaf node $\top$ and one decision node describing all possible values of features A and B, i.e. $(\mathtt{Min} \leq \mathtt{A} \leq \mathtt{Max}) \wedge (\mathtt{Min} \leq \mathtt{B} \leq \mathtt{Max})$, thus representing the uninitialized store for all variants. Analysis elements are propagated forward from the initial to the final location taking assignments and tests into account with delayed widening around while. Hence, at location ② we infer a decision tree with the leaf node: $(\mathtt{y} = 0 \wedge \mathtt{A} = \mathtt{x})$, and the same decision node as in location ①. This decision node is split by the test at location ⓗ, giving rise to the node $(\mathtt{A} \leq \mathtt{B})$, where the left-subtree represents the control flow that executes while zero times (so we propagate leaf node at loc. ② without changing it here), whereas the right-subtree represents the control flow that executes while one or more times. Hence, the decision tree inferred at the location ⑤ is shown in Fig. 1. Notice that the inner nodes of the decision tree in Fig. 1 are labeled with *polyhedral* linear constraints defined over feature variables A and B, while the leaves are labeled with *polyhedral* linear constraints defined over program and feature variables x, y, A and B. The edges of decision trees are labeled with the truth value of the decision on the parent node: we use solid edges for true (i.e., the constraint in the parent node is satisfied) and dashed edges for false (i.e., the negation of the constraint in the parent node is satisfied). Hence, this decision tree represents the following disjunctive property in first-order logic:

$$(\mathtt{A} \leq \mathtt{B} \wedge \mathtt{y} = 0 \wedge \mathtt{A} = \mathtt{x}) \vee (\mathtt{A} - \mathtt{B} \geq 1 \wedge \mathtt{B} = \mathtt{x} \wedge \mathtt{A} = \mathtt{B} + \mathtt{y})$$

From this property, we can see that the given assertion $(\mathtt{y} > 2)$ may be valid in the leaf node that can be reached along the path satisfying the constraint $\neg(\mathtt{A} \leq \mathtt{B})$, i.e. $(\mathtt{A} - \mathtt{B} \geq 1)$. In fact, $(\mathtt{y} > 2)$ holds when $(\mathtt{A} - \mathtt{B} \geq 1 \wedge \mathtt{B} = \mathtt{x} \wedge \mathtt{A} = \mathtt{B} + \mathtt{y} \wedge \mathtt{y} > 2)$ holds, which is computed using the meet ($\sqcap$) operator from the Polyhedra domain of the corresponding polyhedra constraints $(\mathtt{A} - \mathtt{B} \geq 1 \wedge \mathtt{B} = \mathtt{x} \wedge \mathtt{A} = \mathtt{B} + \mathtt{y})$ and $(\mathtt{y} > 2)$. Note that Polyhedra domain is used as parameter in the given decision tree domain for implementing decision and leaf nodes. This way, we infer that the assertion is valid is when the stronger constraint $(\mathtt{A} - \mathtt{B} \geq 3)$ is satisfied. Thus, any variant that satisfies the above constraint $(\mathtt{A} - \mathtt{B} \geq 3)$ represents a "correct" solution to the Loop1A sketch.

To find a "correct & optimal" solution, we perform a *backward quantitative termination lifted analysis* based on decision trees [23] of the Loop1A sub-family satisfying $(\mathtt{A} - \mathtt{B} \geq 3)$. It infers piecewise-defined ranking functions represented by decision trees, where decision nodes are linear constraints over program and feature variables and leaf nodes are affine functions over program and feature variables. The affine functions provide an upper bound on the number of execution steps until termination. The backward termination analysis starts at final location, where the input decision tree has one decision node: $(\mathtt{A} - \mathtt{B} \geq 3)$ and one leaf node: 0 (i.e. zero function describing that there are zero execution steps until termination). Analysis elements are then propagated backwards from final towards initial location. By analyzing while-loop using delayed widening [9], we obtain at location ⓗ the following ranking function: 2 when $(\mathtt{x} \leq \mathtt{B})$ since in this case while is executed zero times so assertion and while tests are only executed; and $3\mathtt{x} - 3\mathtt{B} + 2$ when $(\mathtt{x} > \mathtt{B})$ since in this case while-body is executed $(\mathtt{x} - \mathtt{B})$-times and there are three execution steps (two assignments and one test) in the body of while. After propagating this ranking function through statements at location ① (x is substituted by A), we infer the decision tree shown in Fig. 2. We can see that the ranking function is: $3\mathtt{A} - 3\mathtt{B} + 4$, since the case $(\mathtt{A} \leq \mathtt{B})$ contradicts with $(\mathtt{A} - \mathtt{B} \geq 3)$. We now call the Z3 solver [25] to solve the following linear optimization problem: find values for A and B that *minimizes* the value of ranking function $3\mathtt{A} - 3\mathtt{B} + 4$ over the constraint $(\mathtt{A} - \mathtt{B} \geq 3) \wedge (3\mathtt{A} - 3\mathtt{B} + 4 > 0)$. Minimizing this function gives us values for A and B that are desirable according to the quantitative criterion while satisfying the given assertion. The solution produced by Z3 is: A=3 and B=0 with the minimal objective 13. Therefore, the synthesizer reports this variant, i.e. complete single program where $??_1$=3 and $??_2$=0, as a "correct & optimal" solution to the Loop1A sketch.

We consider an alternative sketch of Loop1A, denoted by Loop1B, in which the assertion in location ⑤ is $(\mathtt{y} < 8)$. The numerical invariant inferred in location ⑤ is the same as for Loop1A as shown in Fig. 1. However, there are now two solutions to the assertion $(\mathtt{y} < 8)$: $(\mathtt{A} \leq \mathtt{B})$ when the left leaf node is reached, and $(1 \leq \mathtt{A} - \mathtt{B} \leq 7)$ when the right leaf node is reached. We perform two backward termination

[1] Note that Min and Max represent some minimal and maximal representable integers. E.g., we may take $\mathtt{Min} = 0$ and $\mathtt{Max} = 31$ for 5-bit sizes of holes.

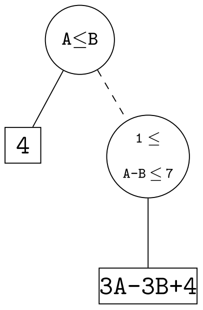


Fig. 3. Lifted ranking function at location ① of Loop1B.

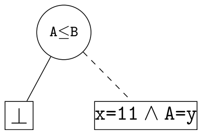


Fig. 4. Lifted numerical invariant at location ⑤ of Loop2A.

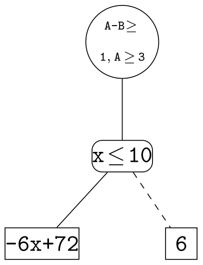


Fig. 5. Lifted ranking function at location ① of Loop2A.

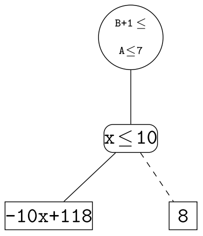


Fig. 6. Lifted ranking function at location ① of Loop2B.

lifted analysis to find optimal solutions for both correct sub-families: $(A \le B)$ and $(1 \le A - B \le 7)$. The lifted ranking function inferred at the initial location is given in Fig. 3. The solutions to the given optimization problem produced by Z3 solver are: A=0, B=0 with the minimal objective 4 for the case $(A \le B)$; and A=1, B=0 with the minimal objective 7 for the case $(1 \le A - B \le 7)$. The solution A=0, B=0 is reported as "correct & optimal" of the Loop1B sketch.

We now illustrate a particular quantitative cost-sensitive (termination) analysis. Let us consider the following Loop2A sketch:

```
void   main(int x) {
①        int y := ??₁;
②        while(x ≤ 10) {
③          if(y > ??₂) x := x+1;
④          else x := x−1; }
⑤        assert(y > 2); //assert(y < 8); }
```

We transform the Loop2A sketch to a program family with two numerical features A and B corresponding to holes $??_1$ and $??_2$ (see Fig. 8). First, we perform a forward numerical lifted analysis, and the lifted numerical invariant inferred at location ⑤ is shown in Fig. 4. The initial decision node $(\mathtt{Min} \le A \le \mathtt{Max}) \wedge (\mathtt{Min} \le B \le \mathtt{Max})$ is split by the test at location ③, giving rise to the node $(A \le B)$, where the left-subtree represents the control flow that executes else statement at location ③ whereas the right-subtree represents the control flow that executes then statement at location ③. From the inferred invariant at location ⑤, we can see that the assertion $(y > 2)$ is valid for variants satisfying: $(A - B \ge 1) \wedge (3 \le A \le \mathtt{Max})$. Suppose we wish to analyze the "correct" sub-family in an environment where there is a significant overhead in the operations: *assignment* and *reads* (dereferencing) of variables because of memory access. In our particular cost model, the overhead of assignment is 2 units and that of dereferencing is 1 unit.[2] Then, we perform a *backward quantitative cost-sensitive (termination) lifted analysis* of the Loop2A "correct" sub-family. The lifted ranking function inferred for this sub-family is shown in Fig. 5. It represents a piecewise-defined ranking function since it depends on the value of the input variable x. To represent graphically piecewise-defined ranking functions in decision trees, we use rounded rectangles to represent second-level decision nodes that are labeled with linear constraints defined over both feature and program variables. Thus, they partition the configuration and memory space, i.e. the possible values of feature and program variables (see Fig. 5). The obtained "correct & optimal" solution is: A=3 and B=0 with the minimal objective 6 when $(x > 10)$ and $-6x + 72$ when $(x \le 10)$. Note that when $(x > 10)$ the while-body is not executed, thus there are two assignments (in locations ① and ②) and two reads of variables (in locations ② and ③) and so the ranking function is 6. When $(x = 10)$ the while-body is executed once, thus there are one assignment and four reads of variables more than the case when $(x > 10)$ and so the ranking function is 12.

Similarly, we can resolve the Loop2B sketch, where we consider the assertion $(y < 8)$ and a cost model with overheads of 2 units for both assignment and dereferencing. The "correct" variants satisfy: $(A - B \ge 1) \wedge (\mathtt{Min} \le A \le 7)$, and the "correct & optimal" solution is (see the lifted ranking function in Fig. 6): A=1 and B=0 with the minimal objective 8 when $(x > 10)$ and $-10x + 118$ when $(x \le 10)$. Note that, the inferred ranking functions for "correct" sub-families of Loop2A and Loop2B in Figs. 5 and 6 do not depend on feature variables, so any "correct" solution is "optimal" as well.

We can see that decision trees inferred by performing lifted analyses of our motivating examples use only one or two leaf nodes (invariants or ranking functions). In contrast, if we use the brute-force enumeration then we will need $(\mathtt{Max} - \mathtt{Min} + 1)^2$ invariants and ranking functions, i.e. one for each variant. This possibility for sharing of equivalent analysis information corresponding to different variants confirms that decision trees are symbolic and compact representation of lifted analysis elements. This is the key for obtaining efficient lifted analyses of program families with large number of variants, and thus for efficiently solving the quantitative sketching problem.

3. Transforming sketches to program families

We now introduce the IMP language that we use to illustrate our work. We describe two extensions of IMP: $\mathrm{IMP}_{??}$ for writing program sketches, and $\overline{\mathrm{IMP}}$ for writing program families. Finally, we define the transformation of sketches to program families and show its correctness. $\overline{\mathrm{IMP}}$ language for program families is specifically designed intermediate language used for constructing suitable analysis and synthesis algorithms for sketches. Thus, $\overline{\mathrm{IMP}}$ is more expressive than the language for classical program families [27] in which features occur only in presence conditions of #if-s. Here, $\overline{\mathrm{IMP}}$ allows features to occur in arbitrary expressions as well.

Note that the intermediate language $\overline{\mathrm{IMP}}$ could be avoided and we could design analysis and synthesis algorithms that work directly on

[2] The default cost model in standard termination analysis is 1 unit for assignments and 0 units for dereferencing.

program sketches from $\mathrm{IMP}_{??}$, rather than on program families from $\overline{\mathrm{IMP}}$. However, in this work we want to re-use the efficient lifted analysis algorithms developed for program families, and to adapt them for program sketching. In order to make this connection explicit, we use $\overline{\mathrm{IMP}}$ as an intermediate language.

3.1. IMP

We use a simple imperative language, called IMP [9,28], for writing general-purpose single-programs. The program variables *Var* are statically allocated, and the only data type is the set $\mathbb{Z}$ of mathematical integers. The syntax of the language is given by:

$$s ::= \mathtt{skip} \mid \mathtt{x} := ae \mid s; s \mid \mathtt{if}\,(be)\,\mathtt{then}\, s\, \mathtt{else}\, s \mid \mathtt{while}\,(be)\,\mathtt{do}\, s \mid \mathtt{assert}\,(be),$$
$$be ::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be,$$
$$ae ::= n \mid [n, n'] \mid \mathtt{x} \mid ae \oplus ae$$

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, $\mathtt{x}$ over program variables *Var*, $\oplus \in \{+, -, *, /\}$, and $\bowtie \in \{<, \leq, =, \neq\}$. Intervals $[n, n']$ denote a random choice of an integer in the interval. The set of all statements s is denoted by *Stm*; the set of all boolean expressions be is denoted by *BExp*; and the set of all arithmetic expressions ae is denoted by *AExp*. Notice that IMP is only used for presentational purposes as a well established minimal language. The introduced methodology is not limited to IMP or its constructs. In fact, our implementation described in Section 7 supports a subset of the C language, which is sufficient to handle realistic programs.

A *program state* $\sigma : \Sigma = Var \to \mathbb{Z}$ is a mapping from program variables to values. The meaning of arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$ is a function from a state to a set of values:

$$[\![n]\!]\sigma = \{n\}, \;\; [\![[n, n']]\!]\sigma = \{n, \ldots, n'\}, \;\; [\![\mathtt{x}]\!]\sigma = \{\sigma(\mathtt{x})\},$$
$$[\![ae_0 \oplus ae_1]\!]\sigma = \{n_0 \oplus n_1 \mid n_0 \in [\![ae_0]\!]\sigma, n_1 \in [\![ae_1]\!]\sigma\}$$

The semantics of boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\mathrm{true}, \mathrm{false}\})$ is the set of possible truth values for expression be in a given state.

$$[\![ae_0 \bowtie ae_1]\!]\sigma = \{n_0 \bowtie n_1 \mid n_0 \in [\![ae_0]\!]\sigma, n_1 \in [\![ae_1]\!]\sigma\}$$
$$[\![\neg be]\!]\sigma = \{\neg t \mid t \in [\![be]\!]\sigma\},$$
$$[\![be_0 \wedge be_1]\!]\sigma = \{t_0 \wedge t_1 \mid t_0 \in [\![be_0]\!]\sigma, t_1 \in [\![be_1]\!]\sigma\}$$
$$[\![be_0 \vee be_1]\!]\sigma = \{t_0 \vee t_1 \mid t_0 \in [\![be_0]\!]\sigma, t_1 \in [\![be_1]\!]\sigma\}$$

The meaning of statements $[\![s]\!] : \Sigma \to \mathcal{P}(\Sigma)$ is the set of final states that can be derived by executing s from some initial input state [9,28].

3.2. $\mathrm{IMP}_{??}$

The language for sketches $\mathrm{IMP}_{??}$ is obtained by extending arithmetic expressions of IMP with a basic integer hole construct, denoted by ??. The numerical hole ?? is a placeholder that the synthesizer must replace with a suitable integer constant.

$$ae ::= \ldots \mid ??$$

Each hole occurrence in a program sketch is assumed to be uniquely labeled as $??_i$ and has a bounded integer domain $[n, n']$. We will sometimes write $??_i^{[n,n']}$ to make explicit the domain of a given hole.

Let H be a set of holes in a program sketch. We define a *control function* $\phi : \Phi = H \to \mathbb{Z}$ to describe the value of each hole in the sketch. Thus, ϕ fully describes a candidate solution to the sketch (i.e. a sketch realization). We write s^{ϕ} to describe a *candidate solution* to the sketch s fully defined by control function ϕ.

3.3. $\overline{\mathrm{IMP}}$

Let $\mathcal{F} = \{A_1, \ldots, A_n\}$ be a finite and totally ordered set of *numerical features* available in a program family. For each feature $A \in \mathcal{F}$, $\mathrm{dom}(A) \subseteq \mathbb{Z}$ denotes the set of possible values that can be assigned to A. A valid combination of feature's values represents a *configuration* k, which specifies one *variant* of a program family. It is given as a *valuation function* $k : \mathcal{F} \to \mathbb{Z}$, which is a mapping that assigns a value from $\mathrm{dom}(A)$ to each feature $A \in \mathcal{F}$, i.e. $k(A) \in \mathrm{dom}(A)$. We assume that only a subset $\mathbb{K}$ of all possible configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be represented by a formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_n = k(A_n))$. The set of configurations $\mathbb{K}$ can be also represented as a formula: $\vee_{k \in \mathbb{K}} k$. We define *feature expressions*, denoted *FeatExp*($\mathcal{F}$), as the set of propositional logic formulas over constraints of $\mathcal{F}$ generated by:

$$\theta ::= \mathrm{true} \mid e_{\mathcal{F}} \bowtie e_{\mathcal{F}} \mid \neg\theta \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2, \qquad e_{\mathcal{F}} ::= n \mid A \mid e_{\mathcal{F}} \oplus e_{\mathcal{F}}$$

where $A \in \mathcal{F}$, $n \in \mathbb{Z}$, $\oplus \in \{+, -, *\}$, and $\bowtie \in \{<, \leq, =, \neq\}$. When a configuration $k \in \mathbb{K}$ satisfies a feature expression $\theta \in FeatExp(\mathcal{F})$, we write $k \models \theta$, where $\models$ is the standard satisfaction relation. We write $[\![\theta]\!]$ to denote the set of configurations from $\mathbb{K}$ that satisfy θ, that is, $k \in [\![\theta]\!]$ iff $k \models \theta$.

The language for program families $\overline{\mathrm{IMP}}$ is obtained by extending IMP with a new compile-time conditional statement for encoding multiple variants and a new basic arithmetic expression construct that represents a feature variable. The new statement "#if (θ) s #endif" contains a feature expression $\theta \in FeatExp(\mathcal{F})$ as a *presence condition*, such that only if θ is satisfied by a configuration $k \in \mathbb{K}$ the statement s will be included in the variant corresponding to k. The syntax is:

$$s ::= \ldots \mid \mathtt{\#if}\,(\theta)\, s\, \mathtt{\#endif}, \qquad ae ::= \ldots \mid A \in \mathcal{F}$$

Any other preprocessor conditional constructs can be desugared and represented only by **#if** construct. For example, #if (θ) s_0 #elif (θ') s_1 #endif is translated into: #if (θ) s_0 #endif ; #if ($\neg\theta \wedge \theta'$) s_1 #endif.

The semantics of $\overline{\mathrm{IMP}}$ has two stages: first, given a configuration $k \in \mathbb{K}$ compute an IMP single-program without #if-s and $A \in \mathcal{F}$; second, the obtained program is evaluated using the standard IMP semantics [14]. The first stage is specified by the projection function P_k, which recursively pre-processes all sub-statements and sub-expressions of statements. Hence, we have $P_k(\mathtt{skip}) = \mathtt{skip}$, $P_k(\mathtt{x} := ae) = \mathtt{x} := P_k(ae)$, $P_k(s; s') = P_k(s); P_k(s')$, and $P_k(ae \oplus ae') = P_k(ae) \oplus P_k(ae')$. For "#if ($\theta$) s #endif", statement s is included in the variant if $k \models \theta$, otherwise, if $k \not\models \theta$ then s is removed.[3]

$$P_k(\mathtt{\#if}\ (\theta)\ s\ \mathtt{\#endif}) = \begin{cases} P_k(s) & \text{if } k \models \theta \\ \mathtt{skip} & \text{if } k \not\models \theta \end{cases}$$

For a feature $A \in \mathcal{F}$, the projection function P_k replaces A with the value $k(A) \in \mathbb{Z}$, that is $P_k(A) = k(A)$.

Example 1. The variant $P_{(A=3)\wedge(B=0)}$(Loop1a) derived from the program family Loop1a in Fig. 7, using configuration $(\mathtt{A} = 3) \wedge (\mathtt{B} = 0)$, is:

$$\mathtt{int\ x} := 3,\ \mathtt{y} := 0;\ \mathtt{while}\,(\mathtt{x} > 0)\ \{\mathtt{x} := \mathtt{x}-1; \mathtt{y} := \mathtt{y}+1;\}\ \mathtt{assert}\,(\mathtt{y} > 2);$$

while the variant $P_{(A=1)}$(vmcai2004a) derived from the program family vmcai2004a in Fig. 9, using configuration $(\mathtt{A} = 1)$, is:

$$\mathtt{int x;\ while}\,(\mathtt{x} \geq 0)\ \mathtt{x} := 1 * \mathtt{x}+10;$$

3.4. Transformation

We now show how to transform an input sketch $\hat{s}$ with a set of m holes $??_1^{[n_1,n_1']}, \ldots, ??_m^{[n_m,n_m']}$ into an output program family $\overline{s}$ with a set of m features $A_1, \ldots, A_m$ with domains $[n_1, n_1'], \ldots, [n_m, n_m']$, respectively. The set of configurations $\mathbb{K}$ in $\overline{s}$ includes all possible combinations of feature values.

If a hole occurs in a (linear) expression that can be exactly represented in the underlying numerical domain $\mathbb{D}$, then we can handle the hole in a more efficient symbolic way by an extended lifted analysis.

[3] Since any $k \in \mathbb{K}$ is a valuation function, we have that either $k \models \theta$ holds or $k \not\models \theta$ (which is equivalent to $k \models \neg\theta$) holds, for any $\theta \in FeatExp(\mathcal{F})$.

```
void main() {
  int x := A, y := 0;
  while (x >B) {
    x := x-1;
    y := y+1; }
  assert (y > 2);
}
```

Fig. 7. The Loop1A family.

```
void main(int x) {
  int y := A;
  while (x  ≤ 10) {
    if (y ≥ B) x := x+1;
    else x := x-1; }
  assert (y > 2);
}
```

Fig. 8. The Loop2A family.

```
void main(){
  int x;
  while (x ≥ 0) {
    #if (A=Min) x:=Min*x+10; #elif ...
    #elif (A=Max-1)x:=(Max-1)*x+10;
    #else x := Max*x+10; ...#endif
  }
}
```

Fig. 9. The vmcai2004A family.

Given the polyhedra domain P, we say that a hole ?? can be *exactly represented* in P, if it occurs in an expression of the form: $\alpha_1 x_1 + \cdots \alpha_i ?? + \cdots \alpha_n x_n + \beta$, where $\alpha_1, \ldots, \alpha_n, \beta \in \mathbb{Z}$ and $x_1, \ldots, x_n$ are program variables or other hole occurrences. Similarly, we define that a hole can be *exactly represented* in the Interval I and the Octagon O domains, if it occurs in expressions of the form: $\pm ?? + \beta$ and $\pm \mathtt{x} \pm ?? + \beta$ (where $\beta \in \mathbb{Z}$, x is a program variable or other hole occurrence).

We now define rewrite rules for eliminating holes ?? from a program sketch $\hat{s}$. Let $s[??^{[n,n']}]$ be a basic (non-compound) statement in which the hole $??^{[n,n']}$ occurs as a sub-expression. When the hole $??^{[n,n']}$ occurs in an expression that can be represented exactly in the numerical domain $\mathbb{D}$, we eliminate ?? using the *symbolic rewrite rule*:

$$s[??^{[n,n']}] \rightsquigarrow s[\mathtt{A}] \tag{SR}$$

Otherwise, if the hole $??^{[n,n']}$ occurs in an expression that cannot be represented exactly in the numerical domain $\mathbb{D}$, we use the *explicit rewrite rule*:

$$s[??^{[n,n']}] \rightsquigarrow \begin{array}{l}\#\mathtt{if}\ (\mathtt{A} = n)\ s[n]\ \#\mathtt{elif} \ldots \#\mathtt{elif}\ (\mathtt{A} = n'-1)\ s[n'-1] \\ \quad \#\mathtt{else}\ s[n']\ \#\mathtt{endif} \ldots \#\mathtt{endif}\end{array} \tag{ER}$$

The set of features $\mathcal{F}$ is also updated with the fresh feature A. If the current sketch $\hat{s}$ being transformed matches any abstract syntax tree (AST) node of the shape $s[??]$, where s ia a basic statement, then we replace $s[??]$ as described by rules (SR) and (ER). We write $\mathtt{Rewrite}(\hat{s})$ to be the resulting program family obtained by repeatedly applying rules (SR) and (ER) on a program sketch $\hat{s}$ to saturation, i.e. until we reach a point when all holes are eliminated. Given a basic statement $s[??]$, we can apply either rule (SR) or rule (ER) depending on whether hole ?? can be exactly represented in a given numerical domain or not. Therefore, this rewriting system is confluent as $\mathtt{Rewrite}(\hat{s})$ will always return the same result (program family) regardless of the order in which we eliminate holes.

Example 2. Reconsider Loop1A and Loop2A sketches from Section 2. All holes ?? can be represented exactly in the Interval domain I, so we use the symbolic (SR) rule to obtain the corresponding program families shown in Figs. 7 and 8. Consider the vmcai2004A sketch taken from SV-COMP:

```
int x; while (x ≥ 0) x := ?? * x+10;
```

The hole ?? occurs in a non-linear expression $?? * \mathtt{x}+10$, so it cannot be represented exactly in any numerical domain $\mathbb{D}$. Thus, we use the explicit (ER) rule to obtain the corresponding program family shown in Fig. 9. □

The following result establishes the correctness of our transformation.

Theorem 3. *Let $\hat{s}$ be a sketch with holes $??_1, \ldots, ??_n$, ϕ be a control function, and $\hat{s}^\phi$ be a candidate solution of $\hat{s}$. Let $\overline{s} = \mathtt{Rewrite}(\hat{s})$ be a program family, in which features $A_1, \ldots, A_n$ correspond to holes $??_1, \ldots, ??_n$. We define a configuration $k \in \mathbb{K}$, s.t. $k(A_i) = \phi(??_i)$ for $1 \le i \le n$. Then, we have: $[\![\hat{s}^\phi]\!] = [\![P_k(\overline{s})]\!]$.*

Proof. By structural induction on statements of sketch $\hat{s}$. The only interesting cases are basic statements $s[??]$ that fulfill requirements of rules (SR) and (ER), since in all other cases we have identity transformations.

For rule (SR), we have:

$$[\![P_k(s[A])]\!] \overset{\text{def. of } P_k}{=} [\![s[k(\mathtt{A})]]\!] \overset{\text{hyp. } k(A)=\phi(??)}{=} [\![s[??]^\phi]\!]$$

For rule (ER), we have:

$$[\![P_k(\#\mathtt{if}\ (\mathtt{A} = n)\ s[n]\ \#\mathtt{elif} \ldots \#\mathtt{elif}\ (\mathtt{A} = n'-1)\ s[n'-1]\ \#\mathtt{else}\ s[n'] \ldots \#\mathtt{endif})]\!]$$
$$\overset{\text{def. of } P_k}{=} [\![s[k(\mathtt{A})]]\!] \overset{\text{hyp. } k(A)=\phi(??)}{=} [\![s[??]^\phi]\!] \quad \square$$

4. Decision tree-based lifted analyses

In the context of program families, *lifting* means taking a static analysis that works on IMP single-programs, and transforming it into an analysis that works on $\overline{\text{IMP}}$ program families, without preprocessing them. In this work, we will use *lifted versions* of the (forward) numerical analysis [9] and the (backward) termination analysis [22,23] from the abstract interpretation framework [8]. They will be used to infer numerical invariants and piecewise-defined ranking functions in all program locations. We work with lifted analyses based on the lifted domain of decision trees [18], in which the leaf nodes belong to an existing single-program domain (e.g., a numerical or termination domain) and decision nodes are linear constraints over feature variables. This way, we encapsulate the set of configurations $\mathbb{K}$ into decision nodes where each top-down path represents a subset of configurations from $\mathbb{K}$, and we store in each leaf node the analysis property generated from the variants corresponding to the given configurations. We now describe the decision tree lifted domain.

4.1. Abstract domain for decision nodes

The domain of decision nodes $\mathbb{C}_{\mathbb{D}_V}$ is the finite set of linear constraints defined over a set of variables $V = \{X_1, \ldots, X_l\}$. $\mathbb{C}_{\mathbb{D}_V}$ is constructed using the numerical domain $\mathbb{D}$ (see Section 4.2.1) by mapping a conjunction of constraints from $\mathbb{D}$ to a finite set of constraints from $\mathbb{C}_{\mathbb{D}_V}$. From now on, we omit to write subscript V in $\mathbb{C}_{\mathbb{D}_V}$ whenever it is clear from the context.

We assume the set of variables $V = \{X_1, \ldots, X_l\}$ to be a finite and totally ordered, such that the ordering is $X_1 > \cdots > X_k$. We impose a total order $<_{\mathbb{C}_{\mathbb{D}}}$ on $\mathbb{C}_{\mathbb{D}}$ to be the lexicographic order on the coefficients $\alpha_1, \ldots, \alpha_k$ and constant α_{k+1} of the linear constraints, such that:

$$(\alpha_1 \cdot X_1 + \ldots + \alpha_k \cdot X_k + \alpha_{k+1} \geq 0) \ <_{\mathbb{C}_{\mathbb{D}}} \ (\alpha'_1 \cdot X_1 + \ldots + \alpha'_k \cdot X_k + \alpha'_{k+1} \geq 0)$$
$$\iff \ \exists j > 0. \forall i < j. (\alpha_i = \alpha'_i) \wedge (\alpha_j < \alpha'_j)$$

The negation of linear constraints is formed as: $\neg(\alpha_1 X_1 + \ldots \alpha_k X_k + \beta \geq 0) = -\alpha_1 X_1 - \cdots - \alpha_k X_k - \beta - 1 \geq 0$. For example, the negation of $X - 3 \geq 0$ is $-X + 2 \geq 0$. To ensure canonical representation of decision trees, a linear constraint c and its negation $\neg c$ cannot both appear as decision nodes. Thus, we only keep the largest constraint with respect to $<_{C_{\mathbb{D}}}$ between c and $\neg c$.

4.2. Abstract domain for leaf nodes

We assume the existence of a single-program abstract domain $\mathbb{A}$ defined over a set of variables $V = \{X_1, \ldots, X_l\}$. The domain $\mathbb{A}$ is equipped with sound operators for concretization $\gamma_{\mathbb{A}}$, ordering $\sqsubseteq_{\mathbb{A}}$, join $\sqcup_{\mathbb{A}}$, meet $\sqcap_{\mathbb{A}}$, bottom $\bot_{\mathbb{A}}$, top $\top_{\mathbb{A}}$, widening $\nabla_{\mathbb{A}}$, and narrowing $\triangle_{\mathbb{A}}$, as well as sound transfer functions for tests (boolean expressions) $\text{FILTER}_{\mathbb{A}}$, forward assignments $\text{F-ASSIGN}_{\mathbb{A}}$, and backward assignments $\text{B-ASSIGN}_{\mathbb{A}}$. More specifically, the concretization function $\gamma_{\mathbb{A}}$ assigns a concrete meaning to each element in the abstract domain $\mathbb{A}$, ordering $\sqsubseteq_{\mathbb{A}}$ conveys the idea of approximation since some analysis results may be coarser than some other results, whereas join $\sqcup_{\mathbb{A}}$ and meet $\sqcap_{\mathbb{A}}$ convey the idea of convergence since a new abstract element is computed when merging control flows. To analyze loops effectively and efficiently, the convergence acceleration operators such as widening $\nabla_{\mathbb{A}}$ and narrowing $\triangle_{\mathbb{A}}$ are used. Transfer functions provide abstract semantics of expressions and statements that operate on the abstract domain $\mathbb{A}$. Thus, they capture the effect of abstractly analyzing expressions and statements in an abstract element (i.e. abstract state). Hence, transfer function $\text{FILTER}_{\mathbb{A}}(a : \mathbb{A}, be : BExp)$ returns an abstract element from $\mathbb{A}$ obtained by restricting a to satisfy the test be; $\text{F-ASSIGN}_{\mathbb{A}}(a : \mathbb{A}, \mathtt{x} := ae : Stm)$ returns an updated version of a by abstractly evaluating $\mathtt{x} := ae$ in it; whereas $\text{B-ASSIGN}_{\mathbb{A}}(b : \mathbb{A}, \mathtt{x} := ae : Stm)$ returns an abstract element from $\mathbb{A}$ such that by abstractly evaluating $\mathtt{x} := ae$ in it produces the abstract element r. Note that parameter a in $\text{F-ASSIGN}_{\mathbb{A}}$ is an invariant in the initial location of $\mathtt{x} := ae$ that needs to be propagated forward, while parameter b in $\text{B-ASSIGN}_{\mathbb{A}}$ is an invariant in the final location of $\mathtt{x} := ae$ that needs to be propagated backwards. These transfer functions are composed to construct abstract analysis of single programs by induction on the syntax. We will sometimes write $\mathbb{A}_V$ to explicitly denote the set of variables V over which domain $\mathbb{A}$ is defined. In this work, we will use domains $\mathbb{A}_{Var}$, $\mathbb{A}_{\mathcal{F}}$, and $\mathbb{A}_{Var \cup \mathcal{F}}$.

4.2.1. Numerical domains

For the forward numerical analysis, we will instantiate $\mathbb{A}$ with some of the known numerical domains $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$, such as Intervals $\langle I, \sqsubseteq_I \rangle$ [8], Octagons $\langle O, \sqsubseteq_O \rangle$ [29], and Polyhedra $\langle P, \sqsubseteq_P \rangle$ [30]. The elements of I are intervals of the form: $\pm X \geq \beta$, where $X \in V, \beta \in \mathbb{Z}$; the elements of O are conjunctions of octagonal constraints of the form $\pm X_1 \pm X_2 \geq \beta$, where $X_1, X_2 \in V, \beta \in \mathbb{Z}$; while the elements of P are conjunctions of polyhedral constraints of the form $\alpha_1 X_1 + \cdots + \alpha_k X_k + \beta \geq 0$, where $X_1, \ldots, X_k \in V, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}$.

We refer to [9] for a precise definition of all abstract operations and transfer functions of intervals, octagons, and polyhedra domains.

4.2.2. Termination domain

For the backward termination analysis, we will instantiate $\mathbb{A}$ with the termination decision tree domain $\mathbb{T}^T(\mathbb{C}_{\mathbb{D}_{Var \cup \mathcal{F}}}, \mathbb{F})$, also written $\mathbb{T}^T$ for short, introduced by Urban and Mine [22,23], where $\mathbb{C}_{\mathbb{D}_{Var \cup \mathcal{F}}}$ is the domain for decision nodes and $\mathbb{F}$ is the domain of *affine functions* for leaf nodes. The elements of $\mathbb{F}$ are:

$$\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\} \cup \{f : \mathbb{Z}^{|Var \cup \mathbb{F}|} \rightarrow \mathbb{N} \mid f(x_1, \ldots, x_n) = m_1 x_1 + \cdots + m_n x_n + q\}$$

where $f \in \mathbb{F}$ is a natural-valued function of program and feature variables representing an upper bound on the number of steps to termination; the element $\bot_{\mathbb{F}}$ represents potential non-termination; and $\top_{\mathbb{F}}$ represents the lack of information to conclude. The function $f \in \mathbb{F}$ represents a piece of a partially-defined ranking function. The leaf nodes belonging to $\mathbb{F} \backslash \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ and $\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ represent *defined* and *undefined* leaf nodes, respectively. The function $\mathtt{succ}_{\mathbb{F}} : \mathbb{F} \rightarrow \mathbb{F}$ increments the constant of a defined function f to take into account that one more execution step is needed before termination:

$$\mathtt{succ}_{\mathbb{F}}(f) = \begin{cases} f & \text{if } f \in \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\} \\ \lambda x_1, \ldots, x_n . f(x_1, \ldots, x_n) + 1 & \text{otherwise} \end{cases}$$

A *termination decision tree* $t' \in \mathbb{T}^T$ is: either a leaf node $\langle\!| f |\!\rangle$ with $f \in \mathbb{F}_A$, or $[\![c' : tl', tr']\!]$, where $c' \in \mathbb{C}_{\mathbb{D}_{Var \cup \mathcal{F}}}$ (denoted by $t'.c$) is the smallest constraint with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ appearing in the tree t', tl' (denoted by $t'.l$) is the left subtree of t' representing its *true branch*, and tr' (denoted by $t'.r$) is the right subtree of t' representing its *false branch*. The path along a decision tree establishes a set of program states and a set of configurations (those that satisfy the encountered constraints), and the leaf nodes represent the partially defined ranking functions over the corresponding program states and configurations.

We use the $\mathtt{succ}_{\mathbb{F}}$ function to define the transfer functions for backward assignment and tests [23] in order to record that one more execution step is needed to termination before executing those operations. The transfer function $\mathtt{B-ASSIGN}_{\mathbb{T}^T}(t', \mathtt{x} := ae)$ substitutes the arithmetic expression ae to the variable $\mathtt{x}$ in linear constraints occurring within decision nodes of t' and in functions occurring in leaf nodes of t', whereas the transfer function $\mathtt{FILTER}_{\mathbb{T}^T}(t', be)$ generates a set of linear constraints J from test be and restricts t' such that all paths satisfy the constraints from J. Finally, both transfer functions increment the constant of defined functions $f \in \mathbb{F} \backslash \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ in all leaf nodes of t' by calling $\mathtt{succ}_{\mathbb{F}}(f)$. We refer to [23] for precise definition of all abstract operations and transfer functions of $\mathbb{T}^T$.

Example 4. Reconsider the variant $P_{(A=3)\wedge(B=0)}(\text{Loop1A})$ from Example 1 derived from the program family Loop1A in Fig. 7 (see also Section 2). The backward termination analysis of $P_{(A=3)\wedge(B=0)}(\text{Loop1A})$ infers the ranking function $[\![(x > 0) : \langle\!| 3x+2 |\!\rangle, \langle\!| 2 |\!\rangle]\!]$ (i.e., $\mathtt{if}\ (x > 0)\ \mathtt{then}\ (3x+2)\ \mathtt{else}\ (2)$) at location ⓗ before the `while`-test. That is, if $(x \leq 0)$ then `while` is executed zero times so there are 2 execution steps by analyzing the assertion and `while`-test; if $(x > 0)$ then `while` is executed x-times and there are 3 execution steps (two assignments and `while`-test) for each `while` iteration, plus there are additional 2 execution steps obtained by analyzing the assertion and `while`-test. Finally, the ranking function 13 is inferred at initial location ①, since there are 2 execution steps to analyze initial assignments $x := 3$, $y := 0$ and by substituting 3 to x we obtain: $3 \cdot 3 + 2 + 2 = 13$.

4.3. Decision tree lifted domains

We now define the decision tree lifted domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{A}_{Var \cup \mathcal{F}})$, written $\mathbb{T}$ for short, for representing lifted analysis properties [18]. A *decision tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ is either a leaf node $\langle\!| a |\!\rangle$ with $a \in \mathbb{A}$, or $[\![c : tl, tr]\!]$, where c, tl, tr are defined as in termination decision trees. The path along a decision tree establishes the set of configurations (those that satisfy the encountered constraints), and the leaf nodes represent the analysis properties for the corresponding configurations.

Operations. The *concretization function* $\gamma_{\mathbb{T}}$ of a decision tree $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ returns $\gamma_{\mathbb{A}}(a)$ for $k \in \mathbb{K}$ that satisfies the set $C \subseteq \mathbb{C}_{\mathbb{D}}$ of constraints accumulated along the top-down path to the leaf node $a \in \mathbb{A}$.

The binary operations rely on the algorithm for *tree unification* [18, 23], which finds a common labeling of decision nodes of two trees t_1 and t_2. Note that the tree unification does not lose any information. All binary operations, including ordering $\sqsubseteq_{\mathbb{T}}$, join $\sqcup_{\mathbb{T}}$, meet $\sqcap_{\mathbb{T}}$, widening $\nabla_{\mathbb{T}}$, and narrowing $\triangle_{\mathbb{T}}$, are performed leaf-wise on the unified decision

Algorithm 1: $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$ **when vars**$(ae) \subseteq Var$

```
1 if isLeaf(t) then return ⟨|ASSIGN_A(t, x:=ae)|⟩;
2 else return
    [[t.c : ASSIGN_T(t.l, x:=ae, C∪{t.c}), ASSIGN_T(t.r, x:=ae, C∪{¬t.c})]]
    ;
```

Algorithm 2: $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$ **when vars**$(ae) \subseteq Var \cup \mathcal{F}$

```
1 if isLeaf(t) then
2     return ASSIGN_{A_{Var∪F}}(t ⊎_{Var∪F} C, x:=ae);
3 else return
    [[t.c : ASSIGN_T(t.l, x:=ae, C∪{t.c}), ASSIGN_T(t.r, x:=ae, C∪{¬t.c})]]
    ;
```

trees. For example, the ordering $t_1 \sqsubseteq_{\mathbb{T}} t_2$ of two unified decision trees t_1 and t_2 is defined recursively as:

$$\langle\!| a_1 |\!\rangle \sqsubseteq_{\mathbb{T}} \langle\!| a_2 |\!\rangle \;=\; a_1 \sqsubseteq_{\mathbb{A}} a_2,$$
$$[\![c : tl_1, tr_1]\!] \sqsubseteq_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = (tl_1 \sqsubseteq_{\mathbb{T}} tl_2) \wedge (tr_1 \sqsubseteq_{\mathbb{T}} tr_2)$$

The top is: $\top_{\mathbb{T}} = \langle\!| \top_{\mathbb{A}} |\!\rangle$, while the bottom is: $\bot_{\mathbb{T}} = \langle\!| \bot_{\mathbb{A}} |\!\rangle$.

Transfer functions. We define lifted transfer functions for analyzing tests, (forward and backward) assignments ($\text{ASSIGN}_{\mathbb{T}}$), and `#if`-s [18]. We consider two types of tests $be \in BExp$ and assignments x $:=ae \in Stm$: when be and ae contain only program variables; and when be and ae contain both feature and program variables.

Transfer function $\text{ASSIGN}_{\mathbb{T}}$.[4] for handling an assignment x $:=ae$ in the input tree t, when x $\in Var$ and the set of variables in ae is $vars(ae) \subseteq Var$, is implemented by applying the corresponding transfer function of the leaf domain $\text{ASSIGN}_{\mathbb{A}}$ leaf-wise, as shown in Algorithm 1. Similarly, transfer function $\text{FILTER}_{\mathbb{T}}$ for handling tests $be \in BExp$, when $vars(be) \subseteq Var$, is implemented by applying $\text{FILTER}_{\mathbb{A}}$ leaf-wise.

Transfer function $\text{ASSIGN}_{\mathbb{T}}$ for x:= ae, when vars$(ae) \subseteq Var \cup \mathcal{F}$, is given in Algorithm 2. It accumulates into the set $C \subseteq \mathbb{C}_{\mathbb{D}}$ (initialized to $\mathbb{K}$) constraints encountered along the paths of the decision tree t (Line 3), up to the leaf nodes in which assignment is performed by $\text{ASSIGN}_{\mathbb{A}_{Var\cup\mathcal{F}}}$. That is, we first merge constraints from the leaf node t defined over $Var \cup \mathcal{F}$ and constraints from decision nodes $C \subseteq \mathbb{C}_{\mathbb{D}_{\mathcal{F}}}$ defined over $\mathcal{F}$, by using $\uplus_{Var\cup\mathcal{F}}$ operator, and then we apply $\text{ASSIGN}_{\mathbb{A}_{Var\cup\mathcal{F}}}$ on the obtained result (Line 2).

Transfer function $\text{FILTER}_{\mathbb{T}}$ for test be, when vars$(be) \subseteq Var \cup \mathcal{F}$, is described by Algorithm 3. Similarly to $\text{ASSIGN}_{\mathbb{T}}$ in Algorithm 2, it accumulates the constraints along the paths in a set $C \subseteq \mathbb{C}_{\mathbb{D}}$ up to the leaf nodes, and applies $\text{FILTER}_{\mathbb{A}_{Var\cup\mathcal{F}}}$ on an abstract element obtained by merging constraints in the leaf node and in C (Line 2). The obtained result a' is a new leaf node, and additionally a' is projected on feature variables using $\restriction_{\mathcal{F}}$ operator to generate a new set of constraints J that is added to the given path to a' by using function call $\texttt{RESTRICT}(\langle\!| a' |\!\rangle, C, J \backslash C)$ (Lines 3–5). Function $\texttt{isRedundant}(J, C)$ checks if the constraints from J are redundant with respect to C, in which case they are not added to the path to a'. Function $\texttt{RESTRICT}(\langle\!| a |\!\rangle, C, J)$, given in Algorithm 4, takes as input a decision tree in the form of a leaf node $\langle\!| a |\!\rangle$, a set C of constraints, and a set J of linear constraints in canonical form that need to be added to $\langle\!| a |\!\rangle$. At each iteration, the smallest linear constraint j with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ is extracted from J (Line 3), and is added to the decision tree (Line 5) if $\texttt{isRedundant}(\{j\}, C)$ is false.

After applying transfer functions, the obtained decision trees may contain some redundancy that can be exploited to further compress them. We use several optimizations [18]. E.g., if constraints on a path to some leaf are unsatisfiable, we eliminate that leaf node; if a decision

Algorithm 3: $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when vars**$(be) \subseteq Var \cup \mathcal{F}$

```
1 if isLeaf(t) then
2     a' = FILTER_{A_{Var∪F}}(t ⊎_{Var∪F} C, be);
3     J = a' ↾_F;
4     if isRedundant(J, C) then return ⟨|a'|⟩;
5     else return RESTRICT(⟨|a'|⟩, C, J\C);
6 else return
    [[t.c : FILTER_T(t.l, x:=e, C∪{t.c}), FILTER_T(t.r, x:=e, C∪{¬t.c})]]
    ;
```

Algorithm 4: $\texttt{RESTRICT}(\langle\!| a |\!\rangle, C, J)$

```
1 if isEmpty(J) then return ⟨|a|⟩;
2 else
3     j = min_{<C_D}(J);
4     if isRedundant({j}, C) then return
        RESTRICT(⟨|a|⟩, C, J\{j});
5     return [[j : RESTRICT(⟨|a|⟩, C ∪ {j}, J\{j}), ⟨|⊥_A|⟩]]
```

node contains two same subtrees, then we keep only one subtree and we also eliminate the decision node, etc.

4.4. Decision tree-based lifted analysis

Operations and transfer functions of $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ and $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$ are used to perform the numerical and termination lifted analysis of program families, respectively. The *numerical lifted analysis* derived from $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$, written as $\mathbb{T}^F$ for short, is a pure *forward* analysis that infers numerical invariants in all program locations. We define the analysis function $[\![s]\!]^F\, t$ that takes as input a decision tree t corresponding to the initial location of statement s, and outputs a decision tree over-approximating the numerical invariant in the final location of s. The input decision tree $t^{\mathbb{K}}_{in,F}$ at the initial location of a program family has only one leaf node $\top_{\mathbb{D}_{Var\cup\mathcal{F}}}$ and decision nodes that define the set $\mathbb{K}$. Lifted invariants (decision trees) are propagated forward from the initial location towards the final location. The analysis function $[\![s]\!]^F : \mathbb{T} \to \mathbb{T}$ for each statement s is given in Fig. 10. For `#if` $(\theta)\, s$ `#endif`, all leaves of the input tree that correspond to configurations that satisfy θ are updated by the effect of evaluating statement s, while all other leaves are not updated. Note that $\text{FILTER}_{\mathbb{T}}(t, \theta, \mathbb{K})$ is defined by Algorithm 3 since θ contains only feature variables, i.e vars$(\theta) \subseteq \mathcal{F}$. For a `while` loop, $\text{lfp}^F \phi^F$ is the limit of the following increasing chain defined by delayed widening:

$$y_0 = \bot_{\mathbb{T}}; \quad y_{n+1} = \phi^F(y_n), \text{ if } n < N; \quad y_{n+1} = y_n \nabla_{\mathbb{T}} \phi^F(y_n), \text{ if } n \ge N \qquad (1)$$

for some fixed number N denoting the widening delay.

Similarly, we define the *termination lifted analysis* derived from $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$, written as $\mathbb{T}^B$ for short, which propagates decision trees from $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$ in a backward direction [23,31]. It is a pure *backward* analysis that infers ranking functions in all program locations. We define the analysis function $[\![s]\!]^B\, t$ that takes as input a decision tree t in the final location of statement s, and outputs a decision tree over-approximating the ranking function in the initial location of s. The input decision tree $t^{\mathbb{K}}_{in,B}$ at the final location of a program family has only one leaf node 0 (zero function) and decision nodes that define the set $\mathbb{K}$. Lifted ranking functions (decision trees) are propagated backward from the final towards the initial location taking assignments, `#if`-s, and tests into account with delayed widening and narrowing around `while`-s..

We establish correctness of the lifted analysis based on $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ by showing that it produces identical results with the brute-force enumeration approach based on the domain $\mathbb{A}$. Let $[\![s]\!]_{\mathbb{T}}$ denotes the

[4] Note that ASSIGN is an abbreviation for both F-ASSIGN and B-ASSIGN

$$
\begin{array}{|l|}
\hline
[\![\texttt{skip}]\!]^F t = t \\
[\![\texttt{x := } ae]\!]^F t = \texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, \mathbb{K}) \\
[\![s_1 \texttt{ ; } s_2]\!]^F t = [\![s_2]\!]^F([\![s_1]\!]^F t) \\
[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]^F t = [\![s_1]\!]^F(\texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K})) \\
\qquad\qquad\qquad\qquad\qquad \sqcup_{\mathbb{T}} [\![s_2]\!]^F(\texttt{FILTER}_{\mathbb{T}}(t, \neg be, \mathbb{K})) \\
[\![\texttt{while } be \texttt{ do } s]\!]^F t = \texttt{FILTER}_{\mathbb{T}}(\mathrm{lfp}^F \phi^F, \neg be, \mathbb{K}) \\
\phi^F(x) = t \sqcup_{\mathbb{T}} [\![s]\!]^F(\texttt{FILTER}_{\mathbb{T}}(x, be, \mathbb{K})) \\
[\![\texttt{\#if } (\theta)\, s \texttt{ \#endif}]\!]^F t = [\![s]\!]^F \mathrm{FILTER}_{\mathbb{T}}(t,\theta,\mathbb{K}) \sqcup_{\mathbb{T}} \mathrm{FILTER}_{\mathbb{T}}(t,\neg\theta,\mathbb{K}) \\
[\![\texttt{assert } (be)]\!]^F t = \texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K}) \\
\hline
\end{array}
$$

Fig. 10. The (forward) numerical lifted analysis function $[\![s]\!]^F : \mathbb{T} \to \mathbb{T}$.

transfer function of statement s of $\overline{\text{IMP}}$ in $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, while $[\![s]\!]_{\mathbb{A}}$ denotes the transfer function of statement s of IMP in $\mathbb{A}$. Given $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, we denote by $\pi_k(t) \in \mathbb{A}$ the leaf node of tree t that corresponds to the variant $k \in \mathbb{K}$. That is, the constraints along the path to the leaf node $\pi_k(t)$ are satisfied by $k \in \mathbb{K}$. We want to show that lifted analysis of s on domain $\mathbb{T}$ produces the same results as analyzing all variants $P_k(s)$, for all $k \in \mathbb{K}$, on domain $\mathbb{A}$.

Theorem 5 (*Correctness*). *$\pi_k([\![s]\!]_{\mathbb{T}}(t)) = [\![P_k(s)]\!]_{\mathbb{A}}(\pi_k(t))$ for all $k \in \mathbb{K}$.*

Proof. The proof is by induction on the structure of s. We consider the two most interesting cases of assignments and `#if`-s. The other cases are similar.

Case $\texttt{x :=}e$. $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x :=}ae, \mathbb{K})$ applies $\texttt{ASSIGN}_{\mathbb{A}}(a, \texttt{x :=}ae)$ to each leaf a in the tree t. The proof follows by applying $\texttt{ASSIGN}_{\mathbb{A}}$ to the leaf node $\pi_k(t)$ corresponding to the variant k.

Case $\texttt{\#if}\,(\theta)\, s\ \texttt{\#endif}$. Assume that $k \models \theta$. Then the leaf node $\pi_k(t)$ occurs in $\texttt{FILTER}_{\mathbb{T}}(t, \theta, \mathbb{K})$ but not in $\texttt{FILTER}_{\mathbb{T}}(t, \neg\theta, \mathbb{K})$. Thus, the result of $\pi_k([\![\texttt{\#if}\,(\theta)\, s\ \texttt{\#endif}]\!]_{\mathbb{T}}(t))$ is $[\![s]\!]_{\mathbb{A}}(\pi_k(t))$. On the other hand, we have $P_k(\texttt{\#if}\,(\theta)\, s\ \texttt{\#endif}) = s$ and the result is $[\![s]\!]_{\mathbb{A}}(\pi_k(t))$.

Assume that $k \not\models \theta$. Then the leaf node $\pi_k(t)$ occurs in $\texttt{FILTER}_{\mathbb{T}}(t, \neg\theta, \mathbb{K})$ but not in $\texttt{FILTER}_{\mathbb{T}}(t, \theta, \mathbb{K})$. Thus, the result of $\pi_k([\![\texttt{\#if}\,(\theta)\, s\ \texttt{\#endif}]\!]_{\mathbb{T}}(t))$ is $\pi_k(t)$. On the other hand, $P_k(\texttt{\#if}\,(\theta)\, s\ \texttt{\#endif}) = \texttt{skip}$, thus giving $[\![\texttt{skip}]\!]_{\mathbb{A}}(\pi_k(t)) = \pi_k(t)$. □

Example 6. In Figs. 11 and 12 we depict decision trees at locations ② and ⓗ inferred by performing (forward) numerical analysis based on the domain $\mathbb{T}(\mathbb{C}_P, P)$ of the LOOP1A program family (see Section 2 and Fig. 7). In order to enforce convergence of the analysis, we apply the widening operator at the loop head, i.e. at location ⓗ. Observe how the invariant at location ⑤ shown in Fig. 1 is inferred from the invariant at location ⓗ. Recall that right leaves in both trees correspond to the case when `while` is executed one or more times. Thus, the condition of `while`-termination ($\texttt{x} = \texttt{B}$) and the right leaf of tree in ⓗ imply the right leaf of tree in ⑤.

Subsequently, we perform a (backward) lifted termination analysis based on the domain $\mathbb{T}(\mathbb{C}_P, \mathbb{T}^T)$ of the LOOP1A sub-family satisfying ($\texttt{A} - \texttt{B} \geq 3$). Lifted decision trees inferred at locations ⓗ and ① are shown in Figs. 2 and 13, respectively. We can see how by back-propagating the tree at location ⓗ, denoted $t_{ⓗ}$ (see Fig. 13), via assignments `y := 0` and `x := A` at location ①, we obtain the tree at location ①, denoted $t_{①}$ (see Fig. 2). The transfer function $\text{B-ASSIGN}_{\mathbb{T}}(t_{ⓗ}, \texttt{x := A})$ will generate the tree $t_{①}$ where `x` is replaced with `A`. The new decision node ($\texttt{A} \geq \texttt{B} + 1$) and the leaf node with ranking function 2 are eliminated from $t_{①}$ since they are redundant with respect to the root node ($\texttt{A} - \texttt{B} \geq 3$).

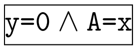


Fig. 11. Invariant at loc. ② of LOOP1A.

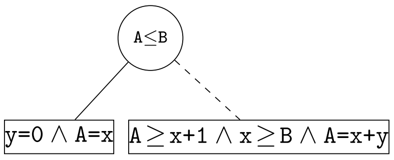


Fig. 12. Invariant at loc. ⓗ of LOOP1A.

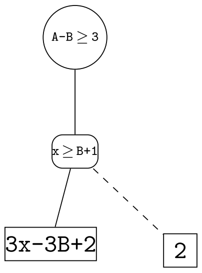


Fig. 13. Ranking fun. at loc. ⓗ of LOOP1A.

5. Quantitative cost-sensitive analyses

The quantitative termination analysis $[\![s]\!]^B$ presented in the previous section establishes an upper bound on the number of execution steps to termination from any location. It uses a default cost model that assigns the cost of one execution step (unit) for skip, assignments, and tests.

However, for any given language there are many quantitative analyses that make sense. They are relative to the resource we want to track and the operational specifics of the architecture on which a program is executed. We can assign different, non-negative costs to each construct and operation in the language. We decorate the cost-sensitive analysis function $[\![s]\!]^B$ with a cost model $\mathcal{M}$ as parameter, which describes the costs assigned to each construct and operation in the language. Suppose we want to analyze our programs in an environment in which there is a significant overhead in the assignment and the dereferencing (reading from variables) operations due to the memory access. We represent this by modeling these operations using analysis functions instrumented with costs incurred during their execution. That is, we attach some non-negative costs to an operation in the corresponding analysis function. We compute the total cost of the above operations using a cost model $\mathcal{M}$ as follows:

$$\mathcal{M}(\texttt{x :=}ae) = \texttt{cost}_{\texttt{asg}} + \texttt{cost}_{\texttt{read}} * |vars(ae)|$$
$$\mathcal{M}(be) = \texttt{cost}_{\texttt{read}} * |vars(be)|$$

where $\mathtt{cost}_{\mathtt{asg}}$ is the cost of executing assignment, $\mathtt{cost}_{\mathtt{read}}$ is the cost of reading from a variable, and $|vars(ae)|$ denotes the number of variables in ae. We define the function $\mathtt{add}_{\mathbb{F}} : \mathbb{F} \times \mathbb{Z} \to \mathbb{F}$, which adds the second parameter to the constant of a defined function in the first parameter. The undefined function in the first parameter is left unaltered. That is, we have:

$$\mathtt{add}_{\mathbb{F}}(f,n) = \begin{cases} f & \text{if } f \in \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\} \\ \lambda x_1,\ldots,x_n.f(x_1,\ldots,x_n)+n & \text{otherwise} \end{cases}$$

The cost-sensitive analysis function, denoted $[\![s]\!]^B(\mathcal{M})$, is the same as $[\![s]\!]^B$ in Section 4.4, except that it propagates the parameter $\mathcal{M}$ in transfer functions $\mathtt{ASSIGN}_{\mathbb{T}^T}(t, \mathtt{x} := ae, \mathcal{M})$ and $\mathtt{FILTER}_{\mathbb{T}^T}(t, be, \mathcal{M})$. $\mathtt{B-ASSIGN}_{\mathbb{T}^T}$ and $\mathtt{FILTER}_{\mathbb{T}^T}$ work as described before in Section 4.2.2, but now they add non-negative constants (costs) $\mathcal{M}(\mathtt{x} := ae)$ and $\mathcal{M}(be)$ to the defined functions in all leaves of the input decision trees by calling $\mathtt{add}_{\mathbb{F}}(f, \mathcal{M}(\mathtt{x} := ae))$ and $\mathtt{add}_{\mathbb{F}}(f, \mathcal{M}(be))$.

6. Synthesis algorithm

We can now solve the quantitative sketching problem using lifted analysis algorithms. More specifically, we delegate the effort of conducting an effective search of all possible sketch realizations to an efficient lifted static analyzer, which combines the forward numerical and the backward quantitative termination analysis. From the numerical invariant inferred using the forward lifted analysis in the final location, we can find a set of variants for which the given assertions are valid. They represent the "correct" sketch realizations. Subsequently, we perform a backward quantitative termination lifted analysis of the "correct" sub-family of variants. From the ranking function inferred using the backward lifted analysis in the initial location, we can find a set of variants for which the minimal fully-defined ranking function is generated. Those variants represent the "correct & optimal" sketch realizations, and are reported as solutions to the quantitative sketching problem.

The synthesis algorithm SYNTHESIZE($\hat{s}$: Stm) for solving a sketch $\hat{s}$ is given in Algorithm 5. First, we transform the program sketch $\hat{s}$ into a program family $\overline{s} = \mathtt{Rewrite}(\hat{s})$ (Line 1). Then, we call function $[\![\overline{s}]\!]^F t^{\mathbb{K}}_{in,F}$ to perform the forward numerical lifted analysis of $\overline{s}$. The inferred decision tree t_F at the final location of $\overline{s}$ is analyzed by function FindCorrect (Line 3) to find the sets of variants for which non-$\bot_{\mathbb{D}}$ and non-$\top_{\mathbb{D}}$ leaf nodes are reachable. The set of variants for which $\bot_{\mathbb{D}}$ leaf node is reachable are "incorrect" with respect to the given assertions; whereas the set of variants for which $\top_{\mathbb{D}}$ leaf node is reachable are "I don't know" (inconclusive), which means that the over-approximations prevent the analyzer from giving a definite answer with respect to the given assertions. For each non-$\bot_{\mathbb{D}}$ and non-$\top_{\mathbb{D}}$ leaf node, we generate the set of variants $\mathbb{K}' \subseteq \mathbb{K}$ that satisfy the encountered linear constraints along the top-down path to that leaf node as well as the given assertions. For each such "correct" set of variants $\mathbb{K}'$, we perform a backward lifted analysis $[\![\overline{s}]\!]^B t^{\mathbb{K}'}_{in,B}$ whose cost model is specified by the quantitative objective we want to observe in the final solution. The inferred decision tree t_B at the initial location of $\overline{s}$ is analyzed by function FindOptimal (Line 6) to find a variant with minimal *defined* leaf nodes from $\mathbb{F}\backslash\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$. The set of variants for which $\bot_{\mathbb{F}}$ leaf node is reachable are "potentially non-terminating"; whereas the set of variants for which $\top_{\mathbb{F}}$ leaf node is reachable are "I don't know" (inconclusive) due to the over-approximations. FindOptimal calls the Z3 solver [25] to solve the following optimization problem: find a model that *minimizes* the value of *fully defined* ranking functions $t' \in \mathbb{T}^T$, such that the linear constraints along the top-down paths to those leaf nodes are satisfied. We say that $t' \in \mathbb{T}^T$ is *fully defined* if all its leaf nodes are defined from $\mathbb{F}\backslash\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$. More formally, given a decision tree $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$, we define the function $\phi[C]t$ that finds a set of pairs (k, t') consisting of valid configurations $k \in \mathbb{K}$ and the corresponding ranking function $t' \in \mathbb{T}^T$ as follows:

Algorithm 5: SYNTHESIZE($\hat{s}$: Stm)

```
1 $\overline{s} = \mathtt{Rewrite}(\hat{s})$;
2 $t_F = [\![\overline{s}]\!]^F t^{\mathbb{K}}_{in,F}$;
3 $C = $ FindCorrect$(t_F)$;
4 while $C \neq \emptyset$ do
5     $\mathbb{K}' = C.remove()$;
6     $t_B = [\![\overline{s}]\!]^B t^{\mathbb{K}'}_{in,B}$;
7     $sol.insert($FindOptimal$(t_B))$
8 return $Min(sol)$
```

$$\phi[C](\langle\!|t'|\!\rangle) = \{(k,t') \mid k \in \mathbb{K}, k \models C, t' \text{ is fully defined}\}$$
$$\phi[C]([\![c : tl, tr]\!]) = \phi[C \cup \{c\}](tl) \cup \phi[C \cup \{\neg c\}](tr)$$

Given a set of configurations $\mathbb{K}$, the required set of pairs (k, t') for a tree $t \in \mathbb{T}$ is obtained by calculating $\phi[\mathbb{K}]t$.

The optimization problem we want to solve is the following. Given a decision tree t_B inferred at the initial location of $\overline{s}$, find a configuration $k \in \mathbb{K}$ such that the corresponding ranking function is minimal. That is,

$$min_{k\in\mathbb{K}}\{t' \mid (k,t') \in \phi[\mathbb{K}]t_B\}$$

The configuration $k \in \mathbb{K}$ (i.e., a mapping from a set of features $\mathcal{F}$ to integers $\mathbb{Z}$) with the minimal ranking function found by Z3 solver is reported as a "correct and optimal" solution to the given quantitative sketching problem.

As standard in abstract analyses, both $[\![\overline{s}]\!]^F$ and $[\![\overline{s}]\!]^B$ are sound (over-approximate) with respect to concrete semantics. Hence, the reported solution by SYNTHESIZE($\hat{s}$) also over-approximates the exact solution. However, we can prove its correctness modulo the chosen abstractions in abstract lifted analyses $[\![\overline{s}]\!]^F$ and $[\![\overline{s}]\!]^B$. If we choose more expressive abstractions, we obtain more precise solutions but more slower analyses times.

Theorem 7. *SYNTHESIZE($\hat{s}$) is correct (modulo chosen abstractions) and terminates.*

Proof. Procedure SYNTHESIZE($\hat{s}$) terminates since all steps in it are terminating. The correctness of SYNTHESIZE($\hat{s}$) follows from correctness of Rewrite (see Theorem 3) and correctness of $[\![\overline{s}]\!]^F$ and $[\![\overline{s}]\!]^B$ (see Theorem 5). ☐

Example 8. Reconsider the Loop1a sketch given in Section 2. The corresponding program family Rewrite(Loop1a) is given in Fig. 7. The numerical invariant t_F inferred in the final location is: $[\![(\mathtt{A} \le \mathtt{B}) : \mathtt{y} = 0 \wedge \mathtt{A} = \mathtt{x}, \mathtt{B} = \mathtt{x} \wedge \mathtt{A} = \mathtt{B} + \mathtt{y}]\!]$. The sub-family of "correct" variants found by FindCorrect is $(\mathtt{A} - \mathtt{B} \ \ge \ 3)$. The ranking function t_B inferred in the initial location is: $3\mathtt{A} - 3\mathtt{B} + 4$. Solution of the generated linear optimization problem $min_{\mathtt{A},\mathtt{B}}\{3\mathtt{A} - 3\mathtt{B} + 4 \mid \mathtt{A} - \mathtt{B} \ge 3 \wedge 3\mathtt{A} - 3\mathtt{B} + 4 > 0\}$ by FindOptimal is: A=3, B=0 with the minimal objective 13.

7. Evaluation

We evaluate our approach for program sketching by comparing it with the Brute-Force enumeration approach that analyzes all variants, one by one, and the most popular sketching tool Sketch that represents the state-of-the-art in the field. The evaluation aims to show that we can use our approach to efficiently resolve various program sketches with numerical data types. To do that, we ask the following research questions:

```
void main(unsigned int x){
  int y := 0;
  while (x ≥ 0) {
    x := x-1;
    if (y<??) y := y+1;
    else y := y-1; }
  assert (y ≥ 1);
}
```

Fig. 14. LoopCondA.

```
void main(unsigned int x){
  int s := 0, y := ??₁;
  int x0 := x, y0 := y;
  while (x ≥ 0) {
    x := x-1;
    while (y ≥??₂) {
      y := y-1; s := s+1; }
  } assert (s ≥ x0+y0);
}
```

Fig. 15. NestedLoop.

```
void main(){
  int x := ??;
  while (x > 10) {
    if (x==25) x := 30;
    if (x ≤ 30) x := x-1;
    else x := 20;
  }
}
```

Fig. 16. tap2008a.

```
void main(){
  int x := ??₁, y:=0;
  while (x > 0) {
    x := -??₂*x+10;
    y := y+1;
  }
  assert (y≤2);
}
```

Fig. 17. vmcai2004.

RQ1: How efficient is our approach compared to the Sketch tool and the Brute-Force approach?
RQ2: Can our approach turn some previously infeasible sketching tasks into feasible ones?
RQ3: Can our approach be used to synthesize sketches with respect to various tailor-made quantitative objectives?

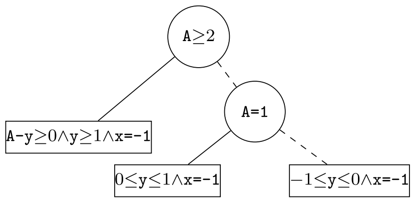
Implementation. We have implemented our synthesis algorithm for quantitative program sketching within the FamilySketcher tool [4]. It uses the lifted decision tree domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, where $\mathbb{A}$ is instantiated either to a numerical abstract domain $\mathbb{D}$ or to the termination decision tree domain $\mathbb{T}^T$. We use the polyhedra domain for $\mathbb{D}$. The abstract operations and transfer functions for the numerical polyhedra domain are provided by the APRON library [24], while for the termination decision tree domain are provided by the Function tool [22]. The tool is written in OCaml and consists of around 7 K LOC. The current front-end of the tool provides a limited support for arrays, pointers, struct and union types. The only basic data type is mathematical integers, which is sufficient for our evaluation.

Experiment setup and benchmarks. Experiments are executed on a 64-bit Intel®Core™ i7-1165G7 CPU@2.80 GHz, VM LUbuntu 20.10, with 8 GB memory, and we use a timeout value of 300 s. We report times needed for the actual static analysis task to be performed. The implementation, benchmarks, and all obtained results are available from: https://github.com/aleksdimovski/Family_sketcher2. We compare our approach with the Sketch version 1.7.6 that uses SAT-based counterexample-guided inductive synthesis [1,2], and with the Brute-Force enumeration approach that analyzes all variants, one by one, using a single-program abstract analysis. The evaluation is performed on several C numerical sketches collected from the Sketch project [1,2], SV-COMP (https://sv-comp.sosy-lab.org/), and from the Syntax-Guided Synthesis Competition (https://sygus.org/) [3]. The chosen benchmarks provide a method to compare the strengths and weaknesses of the given sketching tools. We use the following benchmarks: Loop1a and Loop1b (Section 2), Loop2a and Loop2b (Section 2), LoopCondA and LoopCondB (Fig. 14), NestedLoop (Fig. 15), tap2008a (Fig. 16), and vmcai2004 (Fig. 17).

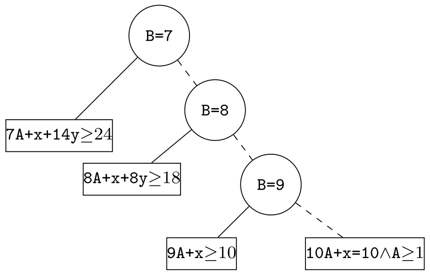
Performance results. We now present the performance results of our empirical study and discuss the implications. Table 1 shows the results of synthesizing our benchmarks. All times are reported as average over five independent executions. The standard deviation of reported results ranges from 0.016 to 0.026 for FamilySketcher, from 0.104 to 0.408 for Sketch, and from 0.082 to 0.105 for Brute-Force. Note that Sketch reports only one "correct" solution for each sketch, which does not have to be "optimal" with respect to the given quantitative objective.

The Loop1a and Loop1b sketches are analyzed using the extended lifted analysis, since both holes are handled symbolically by (SR) rule. Thus, our approach does not depend on sizes of hole domains. FamilySketcher terminates in (around) 0.016 s for Loop1a and in 0.026 s for Loop1b. In contrast, Brute-Force and Sketch do depend on the sizes of holes. Sketch terminates in 37.74 s for 16-bits sizes of holes for Loop1a and in 2.44 s for 16-bits sizes of holes for Loop1b. It times out for bigger sizes of Loop1a. Sketch often reports "correct & optimal" solutions for both sketches. Brute-Force terminates in 21.33 s and 21.38 s for 6-bit sizes of holes for Loop1a and Loop1b, but times out for bigger sizes of holes. Similarly, our tool can handle symbolically Loop2a and Loop2b in 0.060 s and 0.047 s, respectively. However, Sketch cannot resolve them and fails to report a solution, since it uses only 8 unrollments of the loop by default. Note that the number of loop unrollments is a parameter in SKETCH. If the loop is unrolled 11 times, Sketch terminates but often reports the empty trivial solution (addresses **RQ1** and **RQ2**).

The LoopCondA sketch contains one hole ?? (represented by feature A) that can be handled symbolically by (SR) rule, so FamilySketcher has similar running times for all domain sizes. The invariant inferred before the assertion is given in Fig. 18(a). The correct sub-family is $A \geq 2$, and the obtained ranking function for this sub-family is $4x + 8$. Hence, the reported "correct & optimal" solution is $A = 2$. In contrast, Sketch



(a) LoopCondA.



(b) vmcai2004.

Fig. 18. Inferred decision trees at the locations before final assertions.

resolves this example only if the loop is unrolled as many times as is the size of the hole and inputs (e.g., 32 times for 5-bits). Hence, Sketch's performance declines with the growth of size of the hole, and times out for 16-bits (addresses **RQ1**). Consider a variant of LoopCondA sketch, denoted LoopCondB, where one additional hole exists in `while`-test ($x \geq ??_2$). The correct sub-family is $(A \geq 2) \wedge (B \geq x+1)$ (B corresponds to $??_2$). The ranking function for this sub-family is 4, so the "correct & optimal" solution is $(A = 2) \wedge (B = x+1)$. Again, our approach outperforms Sketch and `Brute-Force` on this example (addresses **RQ1**).

The NestedLoop sketch contains two holes that can be handled symbolically by (SR) rule. FamilySketcher terminates in (around) 0.126 s for all sizes of holes. The "correct" solution is $(A - B \geq 0) \wedge (\mathtt{Min} \leq B \leq 1)$, while the "correct & optimal" solution is (A=B=0) and ranking function is 13. On the other hand, `Brute-Force` takes 65.03 s for 5-bit size of holes and times out for larger sizes, while Sketch cannot resolve this benchmark (addresses **RQ1** and **RQ2**).

The sketch tap2008a contains one hole ?? (corresponding feature A) that can be handled symbolically by (SR) rule. Hence, our decision tree-based synthesis approach has similar running times for all domain sizes, reporting as "correct" solutions the constraints: $(\mathtt{Min} \leq A \leq \mathtt{Max})$. The backward lifted analysis finds the ranking function 3 when $(\mathtt{Min} \leq A \leq 10)$, $4A - 37$ when $(11 \leq A \leq 24)$, and 47 when $(31 \leq A)$. Hence, the "correct & optimal" solution is $A = \mathtt{Min}$. In this case, Sketch returns a correct but not optimal solution (addresses **RQ1**).

The vmcai2004 sketch contains two holes. The first one $??_1$ is handled symbolically by (SR) rule while the second one $??_2$ explicitly by (ER) rule. The performance of FamilySketcher depends on the size of $??_2$. The decision tree inferred in the location before the assertion contains one leaf node for each possible value of feature B (features A and B represent $??_1$ and $??_2$). For example, the decision tree when $dom(B) = [7, 10]$ is shown in Fig. 18(b). The sub-family of "correct" solutions is: $(1 \leq A \leq \mathtt{Max}) \wedge (B \geq 10)$, while the "correct & optimal" solution is $(A = 1) \wedge (B = 10)$ with ranking function 6. We observe the exponential growth of running time in this case. Sketch being based on constructing and performing SAT queries is much more successful. It outperforms our approach, but reports only one "correct" solution. However, FamilySketcher still outperforms the `Brute-Force` (addresses **RQ1**). Our approach can achieve better performances if some more expressive numerical abstract domains are employed in the future so that $??_2$ is handled symbolically by (SR) rule.

Cost-sensitive analysis. Reconsider the LoopCondA sketch in Fig. 14, in which the assertion is $(y \geq 0)$ and the `else` branch of `if` is `y := 0`. Thus, there is no reading of variables in it. Now, the correct sub-families are $A \geq 2$ and $A = 1$. We can resolve it using different cost-sensitive analyses. For example, when the cost of assignment is 2 and the cost of reading variables is 0 we obtain the ranking function $6x + 10$ and optimal solution $A = 2$, whereas when the cost of assignment is 1 and the cost of reading variables is 2 the ranking function is $4x + 10$ and optimal solution $A = 1$ (addresses **RQ3**).

We have performed a cost-sensitive analysis of all benchmarks in Table 1 using the cost model with $\mathtt{cost}_{\mathtt{asg}} = 1$ and $\mathtt{cost}_{\mathtt{read}} = 2$. Since there are two assignments and three (resp., four) reads from variables in the body of `while` in Loop1a (resp., Loop2a), the obtained lifted ranking function is $8A - 8B + 6$ (resp., 5 when $(x > 10)$; and $-7x + 82$ when $(x \leq 10)$). Similarly, we infer the lifted ranking function: 5 when $(A \leq 10)$; $7A - 65$ when $(11 \leq A \leq 24)$; and 83 when $(A > 31)$ for tap2008a. Finally, the "correct & optimal" solution to vmcai2004will have ranking function 12. For all benchmarks, we observe small slowdowns in running times of this cost-sensitive analysis compared to the default cost model (see Table 1) that range from < 1% for FamilySketcher to < 2% for `Brute-Force`. Recall that Sketch cannot find "optimal" solutions with respect to cost models.

Discussion. In summary, we can see that FamilySketcher often outperforms Sketch, especially in case of numerical sketches in which holes occur in expressions that can be exactly represented in the underlying numerical domain. But in case of sketches with holes that need to be handled by (ER) rule the performances of our tool decline, which is the consequence of the need to explicitly consider all possible values of those holes. However, even in this case FamilySketcher scales better than the `Brute-Force` approach. This is due to the fact that `Brute-Force` compiles and executes the fixed-point iterative algorithm once for each variant, while our approach does it once per whole family plus there are still possibilities for sharing. Moreover, FamilySketcher reports the "correct & optimal" solution, whereas Sketch reports the first found "correct" solution.

The performances of FamilySketcher can be improved in several ways. First, this is only a prototype implementation of our approach. Many abstract operations and transfer functions of the decision tree lifted domain can be further optimized. Second, instead of APRON we can use other efficient libraries that support numerical domains, such as ELINA [32]. Finally, by using libraries that support more expressive abstract domains, such as non-linear constraints (e.g., polynomials, exponentials [33]), our tool will benefit and more sketches will be handled by (SR) rule.

Threats to validity. The current tool has only limited support for arrays, pointers, `struct` and `union` types. However, the above features are largely orthogonal to the solution proposed here. In particular, these features complicate the semantics of single-programs and implementation of domains for leaf nodes, but have no impact on the semantics of variability-specific constructs and the decision tree lifted domain we introduce for resolving sketches. We perform lifted analysis of relatively small benchmarks. However, the focus of our approach is to combat the realization search space blow-up of sketches, not their LOC size. So, we expect to obtain similar or better results for larger benchmarks. Originally, program sketching [1,2] has been targeting partial programs that express high-level structure of an implementation but leave holes in some low-level details. Thus, the benchmarks have usually been small and accurate, but some algorithmic details are required. The benchmarks we use here have similar characteristics as well, although

Table 1
Performance results of FamilySketcher vs. Sketch vs. Brute-Force.

Bench.	5 bits			6 bits			16 bits		
	Family Sketcher	Sketch	Brute Force	Family Sketcher	Sketch	Brute Force	Family Sketcher	Sketch	Brute Force
Loop1a	0.016	0.192	4.66	0.017	0.197	21.33	0.017	37.74	timeout
Loop1b	0.026	0.203	4.77	0.026	0.216	21.38	0.027	2.44	timeout
Loop2a	0.060	0.200	8.66	0.060	0.202	42.81	0.061	0.348	timeout
Loop2b	0.047	0.203	8.45	0.047	0.205	36.04	0.049	0.521	timeout
LoopCondA	0.042	0.207	1.19	0.042	0.209	2.56	0.043	timeout	timeout
LoopCondB	0.062	0.205	41.19	0.062	0.223	timeout	0.063	timeout	timeout
NestedLoop	0.126	timeout	65.03	0.126	timeout	timeout	0.128	timeout	timeout
tap2008a	0.027	0.201	0.671	0.027	0.206	1.34	0.027	3.07	timeout
vmcai2004	4.69	0.192	5.12	15.52	0.229	19.12	timeout	0.292	timeout

we consider only benchmarks with numerical data types since they are suitable for our approach based on numerical domains. However, in the future we can investigate applications to larger benchmarks from other domains.

8. Related work

The proposed program sketcher uses numerical abstract domains as parameters, so it can be applied for synthesizing numerical programs with numerical data types. The existing widely-known sketching tool Sketch [1,2], which uses SAT-based counterexample-guided inductive synthesis, is more general and especially suited for synthesizing bit-manipulating programs. However, Sketch reasons about loops by unrolling them, so is very sensitive to the degree of unrolling. Our approach being based on abstract interpretation does not have this constraint, since we use the widening extrapolation operator to handle unbounded loops and an infinite number of execution paths in a sound way. This is stronger than fixing a priori a bound on the number of iterations of loops as in the Sketch tool. Moreover, Sketch works by iteratively generating a finite set of inputs and performing SAT queries to identify values for the holes so that the resulting program satisfies all assertions for the given inputs. Further SAT queries are then used to check if the resulting program is correct on all possible inputs. Hence, Sketch may need several iterations to converge reporting only one solution. On the other hand, our approach needs only one iteration to perform lifted analysis reporting several, and very often all, solutions. This is the key for solving the quantitative sketching problem. That is, the set of all "correct" sketch realizations found by the forward lifted analysis can be further refined with respect to a given quantitative objective by the subsequent backward lifted analysis. Another work for solving a quantitative sketching problem is proposed by Chaudhuri et al. [6]. The quantitative optimum they consider is that the expected output value on probabilistic inputs is minimal [34]. They use smoothed proof search and probabilistic analysis to implement this approach in the Fermat tool built on top of Sketch. In contrast, in this work the quantitative optimum we consider is that the number of execution steps to termination is minimal. Adaptive concretization for Sketch [35] combines the benefits of symbolic search, which encodes the search space as a set of SAT queries, and explicit search, which uses brute force enumeration to synthesize highly-influential unknowns. This is similar to our synthesis algorithm, where the holes occurring in linear expressions are handled symbolically using (SR) rules, while the holes in non-linear expressions are handled explicitly using (ER) rules.

Recently, there have been proposed several works that solve the sketching synthesis problem using product line analysis and verification algorithms. Ceska et al. [36] use a counterexample guided abstraction refinement technique for analyzing product lines to resolve probabilistic PRISM model sketches. It starts from considering an abstraction of all possible sketch realizations and iteratively refines the abstraction by splitting the entire family of realizations into subfamilies. Similar abstraction-refinement algorithm has been used to resolve reactive Promela model sketches using single-system SPIN model checker [37]. The work [4] uses a (forward) numerical lifted analysis based on abstract interpretation to resolve numerical sketches by finding "correct" solutions. We extend here this approach by considering the more general quantitative sketching problem, where we find a solution that is also optimal to the given quantitative objective.

Several lifted analysis based on abstract interpretation have been proposed recently [14,17,18] for analyzing program families with #if-s. Midtgaard et al. [14] have proposed the lifted tuple-based analysis phrased in the abstract interpretation framework, while the work [17] improves the tuple representation by using lifted binary decision diagram (BDD) domains. They are applied to program families with only Boolean features. Subsequently, the lifted decision tree domain has been proposed to handle program families with both Boolean and numerical features [18] and dynamic program families [19]. The above lifted analyses are forward and infer numerical invariants in all program locations. Lifted backward termination analysis for inferring ranking functions of numerical program families has been introduced in [31]. Several other efficient implementations of the lifted dataflow analysis from *the monotone framework* (a-la Kildall) [38] have also been proposed in the SPL community. Brabrand et al. [39] have introduced a tuple-based lifted dataflow analysis, whereas an approach based on using variational data structures (e.g., variational CFGs, variational data-flow facts) [16] have been used for achieving efficient dataflow computation of some real-world systems. Finally, SPL^{LIFT} [15] is an implementation of the lifted dataflow analysis formulated within the IFDS framework, which is a subset of dataflow analyses with certain properties, such as distributivity of transfer functions. Many dataflow static analyses, including numerical analyses considered here, are not distributive and cannot be encoded in IFDS. Specifically designed lifted model checking algorithms for verifying game semantics models of program families has been defined in [40–42].

Decision-tree abstract domains have been used in abstract interpretation community recently [29,43,44]. Segmented decision tree abstract domains have also been defined to enable path dependent static analysis [43,44]. Their elements contain decision nodes that are determined either by values of program variables [43] or by the branch (if) conditions [44], whereas the leaf nodes are numerical properties. Urban and Mine [29] use decision tree-based abstract domains to prove program termination. Decision nodes are labeled with linear constraints that split the memory space and leaf nodes contain affine ranking functions for proving program termination. The APRON library has been developed by Jeannet and Mine [24] to support the application of numerical abstract domains in static analysis. The BDDAPRON library [45] is an extension of APRON which adds the power domain of Boolean formulae and any APRON domain. The lifted decision tree domains used here can be seen as generalizations of BDDAPRON, where in decision nodes can be found arbitrary linear constraints instead of only Boolean variables. The ELINA library [32] represents an another efficient implementation of numerical abstract domains that uses improved algorithms, online decomposition, and other performance optimizations.

9. Conclusion

In this work, we proposed a new approach for synthesis of program sketches, such that the resulting program satisfies the combined boolean and quantitative specifications. We have shown that both reasoning tasks can be accomplished using a combination of forward and backward lifted analysis. In particular, we use the forward numerical and the backward termination lifted analyses parameterized by the numerical abstract domains from the APRON library. The quantitative objective we consider here is the number of execution steps to termination, which we want to minimize. Finally, we call the Z3 SMT solver to find a solution to the obtained optimization problem. We experimentally demonstrate the effectiveness of our approach for generating "correct & optimal" solutions of a variety of C benchmarks.

CRediT authorship contribution statement

Aleksandar S. Dimovski: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing, Investigation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link to my code in the manuscript.

References

[1] A. Solar-Lezama, Program sketching, STTT 15 (5–6) (2013) 475–495, http://dx.doi.org/10.1007/s10009-012-0249-7.

[2] A. Solar-Lezama, R.M. Rabbah, R. Bodík, K. Ebcioglu, Programming by sketching for bit-streaming programs, in: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, ACM, 2005, pp. 281–294, http://dx.doi.org/10.1145/1065010.1065045.

[3] R. Alur, R. Bodík, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: Formal Methods in Computer-Aided Design, FMCAD 2013, IEEE, 2013, pp. 1–8, URL http://ieeexplore.ieee.org/document/6679385/.

[4] A.S. Dimovski, S. Apel, A. Legay, Program sketching using lifted analysis for numerical program families, in: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings, in: LNCS, vol.12673, Springer, 2021, pp. 95–112, http://dx.doi.org/10.1007/978-3-030-76384-8_7.

[5] R. Bloem, K. Chatterjee, T.A. Henzinger, B. Jobstmann, Better quality in synthesis through quantitative objectives, in: Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings, in: LNCS, vol.5643, Springer, 2009, pp. 140–156, http://dx.doi.org/10.1007/978-3-642-02658-4_14.

[6] S. Chaudhuri, M. Clochard, A. Solar-Lezama, Bridging boolean and quantitative synthesis using smoothed proof search, in: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, ACM, 2014, pp. 207–220, http://dx.doi.org/10.1145/2535838.2535859.

[7] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[8] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conf. Record of the Fourth ACM Symposium on POPL, ACM, 1977, pp. 238–252, http://dx.doi.org/10.1145/512950.512973, URL http://doi.acm.org/10.1145/512950.512973.

[9] A. Miné, Tutorial on static inference of numeric invariants by abstract interpretation, Found. Trends Program. Lang. 4 (3–4) (2017) 120–372, http://dx.doi.org/10.1561/2500000034.

[10] S. Apel, D.S. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines - Concepts and Implementation, Springer, 2013.

[11] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[12] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, S. Apel, Preprocessor-based variability in open-source and industrial software systems: An empirical study, Empir. Softw. Eng. 21 (2) (2016) 449–482, http://dx.doi.org/10.1007/s10664-015-9360-1.

[13] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, D. Beyer, Strategies for product-line verification: Case studies and experiments, in: 35th Inter. Conference on Software Engineering, ICSE '13, 2013, pp. 482–491.

[14] J. Midtgaard, A.S. Dimovski, C. Brabrand, A. Wasowski, Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. 105 (2015) 145–170, http://dx.doi.org/10.1016/j.scico.2015.04.005.

[15] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, M. Mezini, SPL^{lift}: Statically analyzing software product lines in minutes instead of years, in: ACM SIGPLAN Conference on PLDI '13, 2013, pp. 355–364.

[16] A. von Rhein, J. Liebig, A. Janker, C. Kästner, S. Apel, Variability-aware static analysis at scale: An empirical study, ACM Trans. Softw. Eng. Methodol. 27 (4) (2018) 18:1–18:33, http://dx.doi.org/10.1145/3280986.

[17] A.S. Dimovski, Lifted static analysis using a binary decision diagram abstract domain, in: Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019, ACM, 2019, pp. 102–114, http://dx.doi.org/10.1145/3357765.3359518.

[18] A.S. Dimovski, S. Apel, A. Legay, A decision tree lifted domain for analyzing program families with numerical features, in: Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings, in: LNCS, vol.12649, Springer, 2021, pp. 67–86.

[19] A.S. Dimovski, S. Apel, Lifted static analysis of dynamic program families by abstract interpretation, in: 35th European Conference on Object-Oriented Programming, ECOOP 2021, in: LIPIcs, vol.194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 14:1–14:28, http://dx.doi.org/10.4230/LIPIcs.ECOOP.2021.14.

[20] A.S. Dimovski, A binary decision diagram lifted domain for analyzing program families, J. Comput. Lang. 63 (2021) 101032, http://dx.doi.org/10.1016/j.cola.2021.101032.

[21] A.S. Dimovski, S. Apel, A. Legay, Several lifted abstract domains for static analysis of numerical program families, Sci. Comput. Program. 213 (2022) 102725, http://dx.doi.org/10.1016/j.scico.2021.102725.

[22] C. Urban, FuncTion: An abstract domain functor for termination - (competition contribution), in: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015. Proceedings, in: LNCS, vol.9035, Springer, 2015, pp. 464–466, http://dx.doi.org/10.1007/978-3-662-46681-0_46.

[23] C. Urban, Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes) (Ph.D. thesis), École Normale Supérieure, Paris, France, 2015, URL https://tel.archives-ouvertes.fr/tel-01176641.

[24] B. Jeannet, A. Miné, Apron: A library of numerical abstract domains for static analysis, in: Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings, in: LNCS, vol.5643, Springer, 2009, pp. 661–667, http://dx.doi.org/10.1007/978-3-642-02658-4_52.

[25] L.M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings, in: LNCS, vol.4963, Springer, 2008, pp. 337–340, http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[26] A.S. Dimovski, Quantitative program sketching using lifted static analysis, in: Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Proceedings, in: LNCS, vol.13241, Springer, 2022, pp. 102–122, http://dx.doi.org/10.1007/978-3-030-99429-7_6.

[27] A.S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions for lifted analysis, Sci. Comput. Program. 159 (2018) 1–27.

[28] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, Secaucus, USA, 1999.

[29] C. Urban, A. Miné, A decision tree abstract domain for proving conditional termination, in: Static Analysis - 21st International Symposium, SAS 2014. Proceedings, in: LNCS, vol.8723, Springer, 2014, pp. 302–318, http://dx.doi.org/10.1007/978-3-319-10936-7_19.

[30] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Conference Record of the Fifth Annual ACM Symposium on POPL'78, ACM Press, 1978, pp. 84–96, http://dx.doi.org/10.1145/512760.512770.

[31] A.S. Dimovski, Lifted termination analysis by abstract interpretation and its applications, in: GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021, ACM, 2021, pp. 96–109, http://dx.doi.org/10.1145/3486609.3487202.

[32] G. Singh, M. Püschel, M.T. Vechev, Making numerical program analysis fast, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015, ACM, 2015, pp. 303–313, http://dx.doi.org/10.1145/2737924.2738000.

[33] A.R. Bradley, Z. Manna, H.B. Sipma, The polyranking principle, in: Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings, in: LNCS, vol.3580, Springer, 2005, pp. 1349–1361, http://dx.doi.org/10.1007/11523468_109.

[34] K. Chatterjee, T.A. Henzinger, B. Jobstmann, R. Singh, Measuring and synthesizing systems in probabilistic environments, in: Computer Aided Verification, 22nd International Conference, CAV 2010. Proceedings, in: LNCS, vol.6174, Springer, 2010, pp. 380–395, http://dx.doi.org/10.1007/978-3-642-14295-6_34.

[35] J. Jeon, X. Qiu, A. Solar-Lezama, J.S. Foster, Adaptive concretization for parallel program synthesis, in: Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II, in: LNCS, vol.9207, Springer, 2015, pp. 377–394, http://dx.doi.org/10.1007/978-3-319-21668-3_22.

[36] M. Ceska, C. Dehnert, N. Jansen, S. Junges, J. Katoen, Model repair revamped: On the automated synthesis of Markov chains, in: Essays Dedicated to Scott a. Smolka on the Occasion of His 65th Birthday, in: LNCS, vol.11500, Springer, 2019, pp. 107–125, http://dx.doi.org/10.1007/978-3-030-31514-6_7.

[37] A.S. Dimovski, Model sketching by abstraction refinement for lifted model checking, in: SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, 2022, ACM, 2022, pp. 1845–1848, http://dx.doi.org/10.1145/3477314.3507170.

[38] G.A. Kildall, A unified approach to global program optimization, in: Conf. Record of the ACM Symp. on Principles of Programming Languages, POPL'73, 1973, pp. 194–206, http://dx.doi.org/10.1145/512927.512945.

[39] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, P. Borba, Intraprocedural dataflow analysis for software product lines, T. Aspect-Orient. Softw. Dev. 10 (2013) 73–108.

[40] A.S. Dimovski, Program verification using symbolic game semantics, Theoret. Comput. Sci. 560 (2014) 364–379, http://dx.doi.org/10.1016/j.tcs.2014.01.016.

[41] A.S. Dimovski, Probabilistic analysis based on symbolic game semantics and model counting, in: Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, in: EPTCS, vol.256, 2017, pp. 1–15, http://dx.doi.org/10.4204/EPTCS.256.1.

[42] A.S. Dimovski, Verifying annotated program families using symbolic game semantics, Theoret. Comput. Sci. 706 (2018) 35–53, http://dx.doi.org/10.1016/j.tcs.2017.09.029.

[43] P. Cousot, R. Cousot, L. Mauborgne, A scalable segmented decision tree abstract domain, in: Time for Verification, Essays in Memory of Amir Pnueli, in: LNCS, vol.6200, Springer, 2010, pp. 72–95, http://dx.doi.org/10.1007/978-3-642-13754-9_5.

[44] J. Chen, P. Cousot, A binary decision tree abstract domain functor, in: Static Analysis - 22nd International Symposium, SAS 2015, Proceedings, in: LNCS, vol.9291, Springer, 2015, pp. 36–53, http://dx.doi.org/10.1007/978-3-662-48288-9_3.

[45] B. Jeannet, Relational interprocedural verification of concurrent programs, in: Seventh IEEE Inter. Conf. on Software Engineering and Formal Methods, SEFM'09, IEEE Computer Society, 2009, pp. 83–92, http://dx.doi.org/10.1109/SEFM.2009.29.