Quantitative Program Sketching using Lifted Static Analysis

Aleksandar S. Dimovski[1] (✉)

Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia
`aleksandar.dimovski@unt.edu.mk`

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 analyze it by the specifically designed lifted analysis algorithms based on abstract interpretation. In particular, we use a combination of forward (numerical) and backward (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 "correct & optimal" sketch realizations. We present a prototype implementation of our approach within the FamilySketcher tool for resolving C sketches with numerical types. We have evaluated our approach on a set of benchmarks, and experimental results confirm the effectiveness of our approach.

Keywords: Quantitative program sketching · Software Product Lines · Abstract Interpretation

1 Introduction

A *sketch* [29,30] is a partial program with missing numerical expressions called *holes* to be discovered by the synthesizer. Previous approaches for program sketching [29,30,17] 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. 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 [2,6]. This is so-called *quantitative sketching problem*.

In this paper, we use *lifted static analysis* based on abstract interpretation [25] for program families (a.k.a. software product lines) [8] to solve this quantitative sketching problem. 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 [8]. At compile-time, a variant of 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` [8,21] are today widely used in open-source projects and industry [21]. By using the proposed transformation from program sketches to program families, we reduce 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 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 variants with minimal ranking function are reported as optimal complete programs that solve the original quantitative sketching problem.

To find the required variants (i.e., the solution to the quantitative sketching problem), we use the specifically designed lifted static analysis algorithms, which efficiently analyze all variants of the program family simultaneously, without generating any of them explicitly [3,24,22,28,19,11,20,16]. Lifted analysis processes the common code base of a program family directly, exploiting the similarities among individual variants to reduce analysis effort. It reports precise analysis results for all variants of the family. In particular, we use an efficient, abstract interpretation-based lifted analysis of program families with numerical features [16], where *sharing* is explicitly possible between equivalent analysis elements corresponding to different variants. This is achieved by using a specialized *decision tree lifted domain* [16] 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 labelled with linear constraints over features, while leaf nodes belong to an existing single-program analysis domain (e.g., some numerical domain [25] or the termination domain [31,32]). The decision trees recursively partition the space of all 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). 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. The backward (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., polyhedra). In those cases, we can design very efficient lifted analysis with extended (improved) transfer functions for assignments and tests.

We have implemented our approach in a prototype program synthesizer, called FamilySketcher [17]. The numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library [23] are used as parameters of the underlying decision trees. FamilySketcher calls the Z3 SMT solver [26] to solve the optimization problem that represents the given quantitative objective. We illustrate this approach for automatic completion of various numerical C sketches from the `Sketch` project [29,30], SV-COMP (https://sv-comp.sosy-lab.org/), and the SyGuS-Competition (https://sygus.org/) [1]. We compare performances of our approach against the most popular sketching tool `Sketch` [29,30] and `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 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; (3) We evaluate our approach and compare its performances with the `Sketch` tool and `Brute-Force` enumeration approach.

2 Motivating Examples

Let us consider the Loop1a sketch taken from SyGuS-Competition [1]:

```
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);  }
```

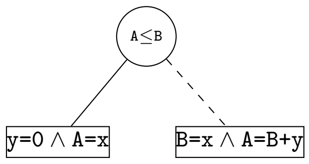

Fig. 1: Lifted numerical invariant at location ⑤ of Loop1a (solid edges = true, dashed edges = false).

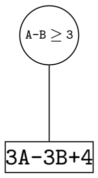

Fig. 2: Lifted ranking function at location ① of Loop1a.

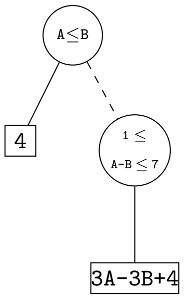

Fig. 3: Lifted ranking function at location ① of Loop1b.

which 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 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 $[\texttt{Min}, \texttt{Max}] \subseteq \mathbb{Z}$. [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. The total number of variants that can be generated from this family is $(\texttt{Max} - \texttt{Min} + 1)^2$, so that each variant corresponds to one possible sketch realization. We perform a *forward numerical lifted analysis* based on decision trees [16] of the Loop1a program family. The decision tree (lifted numerical invariant) 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). Note that linear constraints in decision nodes implicitly take domains of features into account. For example, the decision node $(\texttt{A} \leq \texttt{B})$ is satisfied when $(\texttt{A} \leq \texttt{B}) \wedge (\texttt{Min} \leq \texttt{A} \leq \texttt{Max}) \wedge (\texttt{Min} \leq \texttt{B} \leq \texttt{Max})$. From the invariant inferred at location ⑤ shown in Fig. 1, we can see that the given assertion $(\texttt{y} > 2)$ may be valid in the leaf node that can be reached along the path satisfying the constraint $\neg(\texttt{A} \leq \texttt{B})$, i.e. $(\texttt{A-B} \geq 1)$. In fact, $(\texttt{y} > 2)$ holds when the stronger constraint $(\texttt{A-B} \geq 3)$ is satisfied. Thus, any variant that satisfies the above constraint $(\texttt{A-B} \geq 3)$ represents a "correct" solution to the Loop1a sketch. To find a "correct & optimal" solution,

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

we perform a *backward termination lifted analysis* based on decision trees [13] of the Loop1a sub-family satisfying ($\mathtt{A-B} \geq 3$). The decision tree representing the lifted ranking function of the above sub-family at initial location ① is shown in Fig. 2. [2] Notice that the leaf nodes represent affine functions defined over feature and program variables. We can see that the ranking function is: `3A-3B+4`. We call the Z3 solver [26] to solve the following linear optimization problem: find values for `A` and `B` that *minimizes* the value of ranking function `3A-3B+4` over the constraint $(\mathtt{A-B} \geq 3) \wedge (\mathtt{3A-3B+4} > 0)$. 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. program where $\mathtt{??}_1$=3 and $\mathtt{??}_2$=0, as a "correct & optimal" solution.

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 $(1 \leq \mathtt{A-B} \leq 7)$ when the right leaf node is reached. We perform two backward termination lifted analysis to find optimal solutions for both correct sub-families: $(\mathtt{A} \leq \mathtt{B})$ and $(1 \leq \mathtt{A-B} \leq 7)$. 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 $(\mathtt{A} \leq \mathtt{B})$; and `A=1`, `B=0` with the minimal objective 7 for the case $(1 \leq \mathtt{A-B} \leq 7)$.

Let us consider the Loop2a sketch in Fig. 10. The lifted numerical invariant inferred at location ⑥ is shown in Fig. 4. We can see that the assertion $(\mathtt{y} > 2)$ is valid for variants satisfying: $(\mathtt{A-B} \geq 1) \wedge (3 \leq \mathtt{A} \leq \mathtt{Max})$. 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 labelled 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 3 when $(\mathtt{x} > 10)$ and `-3x+36` when $(\mathtt{x} \leq 10)$. Similarly, we can resolve the Loop2b sketch, where the assertion $(\mathtt{y} < 8)$ is considered. The "correct" variants satisfy: $(\mathtt{A-B} \geq 1) \wedge (\mathtt{Min} \leq \mathtt{A} \leq 7)$, and the "correct & optimal" solution is: `A=1` and `B=0` with the minimal objective 3 when $(\mathtt{x} > 10)$ and `-3x+36` when $(\mathtt{x} \leq 10)$. 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.

From the decision trees inferred by performing lifted analyses of our motivating examples, we can see that the decision tree-based representation uses only one or two leaf nodes, although there are many variants in total. This possibility for sharing of analysis equivalent information corresponding to different variants confirms that decision trees are symbolic and compact representation of lifted

[2] Termination analysis is backward, so the final result is reported in the initial location.

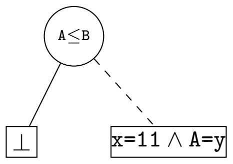

Fig. 4: Lifted numerical invariant before assertion in Loop2a.

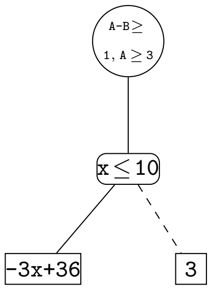

Fig. 5: Lifted ranking function at initial location of Loop2a.

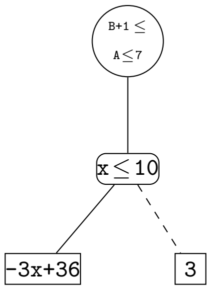

Fig. 6: Lifted ranking function at initial location of Loop2b.

analysis elements. This is the key for obtaining efficient lifted analyses of program families with large configuration spaces, 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.

IMP. We use a simple imperative language, called IMP [27,25], for writing general-purpose single-programs. Program variables *Var* are statically allocated, and the only data type is the set $\mathbb{Z}$ of mathematical integers. Syntax is:

$$\begin{aligned} s &::= \texttt{skip} \mid \texttt{x:=}ae \mid s;s \mid \texttt{if}\,(be)\,\texttt{then}\,s\,\texttt{else}\,s \mid \texttt{while}\,(be)\,\texttt{do}\,s \mid \texttt{assert}\,(be),\\ ae &::= n \mid [n,n'] \mid \texttt{x} \mid ae{\oplus}ae, \qquad be ::= ae{\bowtie}ae \mid \neg be \mid be \land be \mid be \lor be \end{aligned}$$

where n ranges over integers $\mathbb{Z}$, $[n,n']$ over integer intervals, $\texttt{x}$ 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 arithmetic expressions ae is denoted by *AExp*; the set of all boolean expressions be is denoted by *BExp*.

A *program state* $\sigma : \Sigma = Var \to \mathbb{Z}$ is a mapping from program variables to values. The meaning of boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\text{true}, \text{false}\})$, arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$, and statements $[\![s]\!] : \Sigma \to \mathcal{P}(\Sigma)$, are defined by induction on their structure [27,25]. For example, the meaning of an arithmetic expression ae is a function from a state to a set of values:

$$\begin{gathered} [\![n]\!]\sigma = \{n\},\ [\![[n,n']]\!]\sigma = \{n,\ldots,n'\},\ [\![\texttt{x}]\!]\sigma = \{\sigma(\texttt{x})\},\\ [\![ae_0 \oplus ae_1]\!]\sigma = \{n_0 \oplus n_1 \mid n_0 \in [\![ae_0]\!]\sigma, n_1 \in [\![ae_1]\!]\sigma\} \end{gathered}$$

We write $[\![s]\!]$ for the set of final states that can be derived by executing s from some initial input state [27,25].

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

$$ae ::= \ldots \mid \texttt{??}$$

Each hole occurrence in a program sketch is assumed to be uniquely labelled as $\texttt{??}_i$ and has a bounded integer domain $[n, n']$. We will sometimes write $\texttt{??}_i^{[n,n']}$ 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. We write s^{ϕ} to describe a candidate solution to the sketch s fully defined by control function ϕ.

$\overline{\textit{IMP}}$. Let $\mathcal{F} = \{A_1, \ldots, A_n\}$ be a finite and totaly 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 $A \in \mathcal{F}$. We assume that only a subset $\mathcal{K}$ of all possible configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathcal{K}$ can also be represented by a propositional formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_n = k(A_n))$. The set of configurations $\mathcal{K}$ can be also represented as a formula: $\vee_{k \in \mathcal{K}} k$. We define *feature expressions*, denoted *FeatExp*$(\mathcal{F})$, as the set of propositional logic formulas over constraints of $\mathcal{F}$ generated by:

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

When a configuration $k \in \mathcal{K}$ satisfies a feature expression $\theta \in \textit{FeatExp}(\mathcal{F})$, we write $k \models \theta$, where $\models$ is the standard satisfaction relation. We write $[\![\theta]\!]$ to denote the set of configurations from $\mathcal{K}$ 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 arithmetic expression that represents a feature variable. The new statement "`#if` (θ) s `#endif`" contains a feature expression $\theta \in \textit{FeatExp}(\mathcal{F})$ as a *presence condition*, such that only if θ is satisfied by a configuration $k \in \mathcal{K}$ the statement s will be included in the variant corresponding to k. The syntax is:

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

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 the following: `#if` (θ) s_0 `#endif ; #if` $(\neg\theta \wedge \theta')$ s_1 `#endif`. Note that feature variables $\texttt{A} \in \mathcal{F}$ can occur in arbitrary expressions in $\overline{\mathrm{IMP}}$, not only in presence conditions of `#if`-s as in traditional program families [21,24].

The semantics of $\overline{\mathrm{IMP}}$ has two stages: first, given a configuration $k \in \mathcal{K}$ compute an IMP single-program without `#if`-s and $\texttt{A} \in \mathcal{F}$; second, the obtained program is evaluated using the standard IMP semantics [24]. The first stage is specified by the projection function π_k, which recursively pre-processes all sub-statements and sub-expressions of statements. Hence, $\pi_k(\texttt{skip}) = \texttt{skip}$, $\pi_k(\texttt{x:=}ae) = \texttt{x:=}\pi_k(ae)$, $\pi_k(s;s') = \pi_k(s);\pi_k(s')$, $\pi_k(ae \oplus ae') = \pi_k(ae) \oplus \pi_k(ae')$, and $\pi_k(ae \bowtie ae') = \pi_k(ae) \bowtie \pi_k(ae')$. For "$\texttt{#if}\ (\theta)\ s\ \texttt{#endif}$", statement s is included in the variant if $k \models \theta$, otherwise, if $k \not\models \theta$ statement s is removed: [3] $\pi_k(\texttt{#if}\ (\theta)\ s\ \texttt{#endif}) = \begin{cases} \pi_k(s) & \text{if } k \models \theta \\ \texttt{skip} & \text{if } k \not\models \theta \end{cases}$. For a feature $A \in \mathcal{F}$, the projection function π_k replaces A with the value $k(A) \in \mathbb{Z}$, that is $\pi_k(A) = k(A)$.

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

If a hole occurs in a (linear) expression that can be exactly represented in the underlying numerical abstract domain $\mathbb{D}$, then we can handle the hole in a more efficient symbolic way by an extended lifted analysis. 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: $\alpha_1 x_1 + \ldots \alpha_i \texttt{??} + \ldots \alpha_n x_n + \beta$, 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: $\pm\texttt{??} + \beta$ and $\pm\texttt{x} \pm \texttt{??} + \beta$, (where $\beta \in \mathbb{Z}$, x is a program variable or other hole occurrence), respectively.

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

$$s[\texttt{??}^{[n,n']}] \ \rightsquigarrow\ s[\texttt{A}] \qquad \text{(SR)}$$

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

$$s[\texttt{??}^{[n,n']}] \ \rightsquigarrow\ \texttt{#if}\ (\texttt{A=}n)\ s[n]\ \texttt{#elif} \ldots \texttt{#elif}\ (\texttt{A=}n'\texttt{-1})\ s[n'\texttt{-1}]\ \texttt{#else}\ s[n'] \ldots \texttt{#endif} \qquad \text{(ER)}$$

The set of features $\mathcal{F}$ is also updated with the fresh feature A. We write $\texttt{Rewrite}(\hat{s})$ to be the resulting program family obtained by repeatedly applying rules (SR) and (ER) on a program sketch $\hat{s}$ to saturation.

Example 1. Reconsider the Loop1a and Loop2a sketches from Section 2. All holes ?? can be represented exactly in the interval domain, so we use the symbolic (SR) rule to obtain the program family. Consider the sketch: `int x; while (x` $\geq$ 0) `x := ??*x+10`. The hole ?? cannot be represented exactly in any numerical domain $\mathbb{D}$. Thus, we use the explicit (ER) rule to obtain the program family. □

[3] Since any $k \in \mathcal{K}$ 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 $\theta \in \mathit{FeatExp}(\mathcal{F})$.

The following result establishes the correctness of our transformation. It can be proved by structural induction on statements and expressions.

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

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 [25] and the (backward) termination analysis [31] from the abstract interpretation framework [9]. 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 [16], 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 $\mathcal{K}$ into decision nodes where each top-down path represents a subset of configurations from $\mathcal{K}$, and we store in each leaf node the analysis property generated from the variants corresponding to the given configurations.

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 $V = \{X_1, \ldots, X_k\}$. $\mathbb{C}_{\mathbb{D}}$ is constructed using the numerical domain $\mathbb{D}$ (see Section 4.2) by mapping a conjunction of constraints from $\mathbb{D}$ to a finite set of constraints in $\mathcal{P}(\mathbb{C}_{\mathbb{D}})$. We assume the set of variables $V = \{X_1, \ldots, X_k\}$ to be a finite and totally ordered set, such that the ordering is $X_1 > \ldots > X_k$. 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:

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

The negation of linear constraints is formed as: $\neg(\alpha_1 X_1 + \ldots \alpha_k X_k + \beta \geq 0) = -\alpha_1 X_1 - \ldots - \alpha_k X_k - \beta - 1 \geq 0$. 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 $V = \{X_1, \ldots, X_k\}$. 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) FILTER$_{\mathbb{A}}$, forward assignments F-ASSIGN$_{\mathbb{A}}$, and backward assignments B-ASSIGN$_{\mathbb{D}}$. More specifically, FILTER$_{\mathbb{A}}(a : \mathbb{A}, be : \mathit{BExp})$ returns an abstract element from $\mathbb{A}$ obtained by restricting a to satisfy the test be; F-ASSIGN$_{\mathbb{A}}(a : \mathbb{A}, \texttt{x:=}e : \mathit{Stm})$ returns an updated version of a by abstractly evaluating $\texttt{x:=}e$ in it; whereas B-ASSIGN$_{\mathbb{A}}(b : \mathbb{A}, \texttt{x:=}ae : \mathit{Stm})$ returns an abstract element from $\mathbb{A}$ that can lead to the abstract element b to hold after evaluating $\texttt{x:=}ae$. Note that a in F-ASSIGN$_{\mathbb{A}}$ is an invariant in the initial location of $\texttt{x:=}ae$ that needs to be propagated forward, while b in B-ASSIGN$_{\mathbb{A}}$ is an invariant in the final location of $\texttt{x:=}ae$ that needs to be propagated backwards. We will sometimes write $\mathbb{A}_V$ to explicitly denote the set of variables V over which $\mathbb{A}$ is defined. In this work, we will use domains $\mathbb{A}_{\mathit{Var}}$, $\mathbb{A}_{\mathcal{F}}$, and $\mathbb{A}_{\mathit{Var}\cup\mathcal{F}}$.

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 $\langle I, \sqsubseteq_I \rangle$ [9,25], Octagons $\langle O, \sqsubseteq_O \rangle$ [25], and Polyhedra $\langle P, \sqsubseteq_P \rangle$ [25]. 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 $\alpha_1 X_1 + \ldots + \alpha_k X_k + \beta \geq 0$, where $X_1, \ldots X_k \in V, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}$.

For the backward termination analysis, we will instantiate $\mathbb{A}$ with the termination decision tree domain $\mathbb{T}^T(\mathbb{C}_{\mathbb{D}_{\mathit{Var}\cup\mathcal{F}}}, \mathbb{F}_A)$, also written $\mathbb{T}^T$ for short, introduced by Urban and Miné [31,32], where $\mathbb{C}_{\mathbb{D}_{\mathit{Var}\cup\mathcal{F}}}$ is the domain for decision nodes and $\mathbb{F}_A$ is the domain of *affine functions* for leaf nodes. The elements of $\mathbb{F}_A$ are: $\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\} \cup \{f : \mathbb{Z}^{|\mathit{Var}\cup\mathcal{F}|} \to \mathbb{N} \mid f(x_1, \ldots, x_n) = m_1 x_1 + \ldots + m_n x_n + q\}$, where $f \in \mathbb{F}_A$ 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 leaf nodes belonging to $\mathbb{F}_A \backslash \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ and $\{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ represent *defined* and *undefined* leaf nodes, respectively. A *termination decision tree* $t' \in \mathbb{T}^T$ is: either a leaf node $\ll f \gg$ with $f \in \mathbb{F}_A$, or $[\![c' : tl', tr']\!]$, where $c' \in \mathbb{C}_{\mathbb{D}_{\mathit{Var}\cup\mathcal{F}}}$ (denoted by $t'.c$) is the smallest constraint with respect to $<_{\mathbb{D}}$ 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 leaf nodes represent partially-defined ranking functions over the given program states and configurations. The transfer function $\texttt{B-ASSIGN}_{\mathbb{T}^T}(t', \texttt{x:=}ae)$ substitutes the arithmetic expression ae to the variable $\texttt{x}$ in linear constraints occurring within decision nodes of t' and in functions occurring in leaf nodes of t', whereas the transfer function $\texttt{FILTER}_{\mathbb{T}^T}(t', be)$ 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 q of defined functions $f \in \mathbb{F}_A \backslash \{\bot_{\mathbb{F}}, \top_{\mathbb{F}}\}$ in all leaf nodes of t'.

We refer to [25,31] for a precise definition of all operations and transfer functions of intervals, octagons, polyhedra, and termination decision tree domain.

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

```
1 if isLeaf($t$) then return $\ll \texttt{ASSIGN}_{\mathbb{A}}(t, \texttt{x:=}e) \gg$;
2 else return $[\![ t.c : \texttt{ASSIGN}_{\mathbb{T}}(t.l, \texttt{x:=}e, C \cup \{t.c\}), \texttt{ASSIGN}_{\mathbb{T}}(t.r, \texttt{x:=}e, C \cup \{\neg t.c\}) ]\!]$ ;
```

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 [16]. A *decision tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ is either a leaf node $\ll a \gg$ with $a \in \mathbb{A}$, or $[\![c : tl, tr]\!]$, where decision node $c \in \mathbb{C}_{\mathbb{D}}$ (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 the set of configurations (those that satisfy the encountered constraints), and the leaf nodes represent their analysis properties.

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 $k \in \mathcal{K}$ that satisfies the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ 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* [16,31], which finds a common labelling 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 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:

$$\ll a_1 \gg \sqsubseteq_{\mathbb{T}} \ll a_2 \gg = a_1 \sqsubseteq_{\mathbb{A}} a_2, \;\; [\![c : tl_1, tr_1]\!] \sqsubseteq_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = (tl_1 \sqsubseteq_{\mathbb{T}} tl_2) \land (tr_1 \sqsubseteq_{\mathbb{T}} tr_2)$$

The top is: $\top_{\mathbb{T}} = \ll \top_{\mathbb{A}} \gg$, while the bottom is: $\bot_{\mathbb{T}} = \ll \bot_{\mathbb{A}} \gg$.

Transfer functions. We define lifted transfer functions for tests, (forward and backward) assignments ($\text{ASSIGN}_{\mathbb{T}}$), and `#if`-s [16]. We consider several types of tests be and assignments `x:=`ae: 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 the set of variables in ae is $vars(ae) \subseteq Var$, is implemented by applying $\text{ASSIGN}_{\mathbb{A}}$ 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 $\text{vars}(ae) \subseteq Var \cup \mathcal{F}$, is given in Algorithm 2. It accumulates into the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (initialized to $\mathcal{K}$) constraints encountered along the paths of the decision tree t (Line 2), up to the leaf nodes in which assignment is performed by $\text{ASSIGN}_{\mathbb{A}_{Var \cup \mathcal{F}}}$. That is, we first merge

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

```
Algorithm 2: ASSIGN_T(t, x:=ae, C) when vars(ae) ⊆ Var ∪ F
1 if isLeaf(t) then return $\mathtt{ASSIGN}_{\mathbb{A}_{Var\cup\mathcal{F}}}(t \uplus_{Var\cup\mathcal{F}} C, \mathtt{x}{:=}ae)$;
2 else return $[\![t.c : \mathtt{ASSIGN}_{\mathbb{T}}(t.l, \mathtt{x}{:=}e, C\cup\{t.c\}), \mathtt{ASSIGN}_{\mathbb{T}}(t.r, \mathtt{x}{:=}e, C\cup\{\neg t.c\})]\!]$ ;
```

```
Algorithm 3: FILTER_T(t, be, C) when vars(be) ⊆ Var ∪ F
1 if isLeaf(t) then
2     $a' = \mathtt{FILTER}_{\mathbb{A}_{Var\cup\mathcal{F}}}(t \uplus_{Var\cup\mathcal{F}} C, be)$;
3     $J = a' \restriction_{\mathcal{F}}$;
4     if $isRedundant(J, C)$ then return $\ll a' \gg$;
5     else return $\mathtt{RESTRICT}(\ll a' \gg, C, J \backslash C)$;
6 else return $[\![t.c : \mathtt{FILTER}_{\mathbb{T}}(t.l, \mathtt{x}{:=}e, C\cup\{t.c\}), \mathtt{FILTER}_{\mathbb{T}}(t.r, \mathtt{x}{:=}e, C\cup\{\neg t.c\})]\!]$ ;
```

constraints from the leaf node t defined over $Var \cup \mathcal{F}$ and constraints from decision nodes $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}})$ 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 1).

Transfer function $\text{FILTER}_{\mathbb{T}}$ for test be, when $\text{vars}(be) \subseteq Var \cup \mathcal{F}$, is described by Algorithm 3. Similarly to $\text{ASSIGN}_{\mathbb{T}}$ in Algorithm 2, it accumulates the constraints along the paths in a set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ 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 the function `RESTRICT` [16] (Lines 3–5). The function `isRedundant`(J, C) checks if the constraints from J are redundant with respect to the set C.

Finally, transfer function for `#if` directives is defined as:

$$[\![\texttt{\#if } (\theta)\ s\ \texttt{\#end}]\!]_{\mathbb{T}} t = [\![s]\!]_{\mathbb{T}} \text{FILTER}_{\mathbb{T}}(t,\theta,\mathcal{K}) \sqcup_{\mathbb{T}} \text{FILTER}_{\mathbb{T}}(t,\neg\theta,\mathcal{K})$$

where $[\![s]\!]_{\mathbb{T}}(t)$ is transfer function for s and $\text{FILTER}_{\mathbb{T}}(t, \theta, \mathcal{K})$ is defined by Algorithm 3 since θ contains only features. Transfer function for assertions is: $[\![\texttt{assert}(be)]\!]_{\mathbb{T}} = \text{FILTER}_{\mathbb{T}}(t, be, \mathcal{K})$.

After applying transfer functions, the obtained decision trees may contain some redundancy that can be exploited to further compress them. We use several optimizations [16]. E.g., if constraints on a path to some leaf are unsatisfiable, we eliminate that leaf node; if a decision 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 $[\![s]\!]_{\mathbb{T}^F} t$ 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 $t_{in,F}^{\mathcal{K}}$ 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 $\mathcal{K}$. Lifted invariants are propagated forward from the initial location towards the final location taking assignments, `#if`-s, and tests into account with widening and narrowing around `while`-s. We apply delayed widening [9], which means that we start extrapolating by widening after a fixed number of iterations of a loop are analyzed explicitly.

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. It is a pure *backward* analysis that infers ranking functions in all program locations. We define the analysis function $[\![s]\!]_{\mathbb{T}^B} t$ 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 $t_{in,B}^{\mathcal{K}}$ at the final location of a program family has only one leaf node 0 (zero function) and decision nodes that define the set $\mathcal{K}$. Lifted ranking functions are propagated backward from the final towards the initial location.

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 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 $Pr_k(t) \in \mathbb{A}$ the leaf node of tree t that corresponds to the variant $k \in \mathcal{K}$.

Theorem 2. *$Pr_k([\![s]\!]_{\mathbb{T}}(t)) = [\![\pi_k(s)]\!]_{\mathbb{A}}(Pr_k(t))$ for all $k \in \mathcal{K}$.*

Example 2. In Figs. 7 and 8 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). In order to enforce convergence of the analysis, we apply the widening operator at the loop head, i.e. at the location ⓗ before the `while` test. We can see how the invariant at location ⑤ shown in Fig. 1 is inferred from the invariant at location ⓗ.

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 ($\texttt{A-B} \geq 3$). Lifted decision trees inferred at locations ⓗ and ① are shown in Figs. 9 and 2, respectively. We can see how by back-propagating the tree at location ⓗ, denoted $t_{ⓗ}$ (see Fig. 9), 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 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 ($\texttt{A-B} \geq 3$).

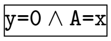

Fig. 7: Invariant at loc. ② of Loop1A.

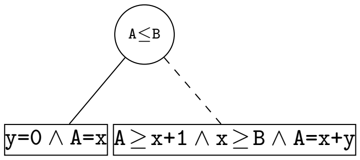

Fig. 8: Invariant at loc. ⓗ of Loop1A.

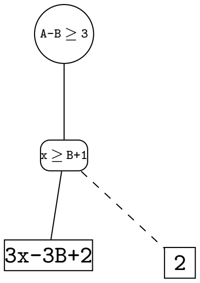

Fig. 9: Ranking fun. at loc. ⓗ of Loop1A.

5 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 termination analyses.

The synthesis algorithm SYNTHESIZE($\hat{s}$: *Stm*) for solving a sketch $\hat{s}$ is given in Algorithm 4. First, we transform the program sketch $\hat{s}$ into a program family $\overline{s} = \texttt{Rewrite}(\hat{s})$ (Line 1). Then, we call function $[\![\overline{s}]\!]_{\mathbb{T}^F} t_{in,F}^{\mathcal{K}}$ 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). For each non-$\bot_{\mathbb{D}}$ and non-$\top_{\mathbb{D}}$ leaf node, we generate the set of variants $\mathcal{K}' \subseteq \mathcal{K}$ 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 $\mathcal{K}$', we perform the backward termination lifted analysis $[\![\overline{s}]\!]_{\mathbb{T}^B} t_{in,B}^{\mathcal{K}'}$. The inferred decision tree t_B is analyzed by function FindOptimal (Line 7). It calls the Z3 solver [26] to solve the following optimization problem: find a model that *minimizes* the value of ranking functions $t' \in \mathbb{T}^T$, such that the linear constraints along the top-down paths to those leaf nodes are satisfied. 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 $k \in \mathcal{K}$ and the corresponding ranking function $t' \in \mathbb{T}^T$ as follows:

$$\phi[C](\ll t' \gg) = \{(k, t') \mid k \in \mathcal{K}, k \models C\}, \;\; \phi[C]([\![c\!:\!tl, tr]\!]) = \phi[C \cup \{c\}](tl) \cup \phi[C \cup \{\neg c\}](tr)$$

The optimization problem is the following. Given a decision tree $t_B \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$ inferred at the initial location of $\overline{s}$, find a configuration $k \in \mathcal{K}$ such that the corresponding ranking function is minimal. That is, $min_{k \in \mathcal{K}} \{t' \mid (k, t') \in \phi[\mathcal{K}]t_B\}$.

The configuration k with the minimal ranking function found by Z3 is reported as a "correct and optimal" solution to the quantitative sketching problem.

Theorem 3. *SYNTHESIZE($\hat{s}$) is correct and terminates.*

Algorithm 4: `SYNTHESIZE`$(\hat{s} : \mathit{Stm})$

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

6 Evaluation

We evaluate our approach for program sketching by comparing it with the `Brute-Force` enumeration approach and the popular `Sketch` tool.

Implementation We have implemented our synthesis algorithm for quantitative program sketching [14] within the FamilySketcher tool [17]. It uses the lifted decision tree domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, where $\mathbb{A}$ is instantiated either to numerical abstract domain $\mathbb{D}$ or to the termination decision tree domain $\mathbb{T}^T$. The abstract operations and transfer functions for the numerical domain $\mathbb{D}$: intervals, octagons, polyhedra, are provided by the APRON library [23], while for the termination decision tree domain are provided by the `Function` tool [32]. The tool is written in OCaml and consists of around 7K LOC. The current 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.

Within the FamilySketcher, we have also implemented the `Brute-Force` enumeration approach that analyzes all variants (sketch realizations), one by one, using the single-program domains $\mathbb{D}$ and $\mathbb{T}^T$.

Experiment setup and Benchmarks All experiments are executed on a 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM LUbuntu 20.10, with 8 GB memory, and we use a timeout value of 300 seconds. All times are reported as average over five independent executions. We report times needed for the actual analysis task to be performed. The implementation is available from [14]: https://zenodo.org/record/5898643#.YhJLRejMLIU. We compare our approach with program sketching tool `Sketch` version 1.7.6 that uses SAT-based counterexample-guided inductive synthesis [30,29], and with the `Brute-Force` enumeration approach. The evaluation is performed on several C numerical sketches collected from the `Sketch` project [30,29], SV-COMP (https://sv-comp.sosy-lab.org/), and the SyGuS-Competition [1]. We use the following benchmarks: Loop1a and Loop1b (Sec. 2), Loop2a and Loop2b (Fig. 10), LoopCond (Fig. 11), NestedLoop (Fig. 12), vmcai2004 (Fig. 13).

Performance Results Table 1 shows the results of synthesizing our benchmarks. Note that `Sketch` reports only one "correct" solution for each sketch, which

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

Fig. 10: Loop2a.

```
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. 11: LoopCond.

```
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. 12: NestedLoop.

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

Fig. 13: vmcai2004.

does not have to be "optimal" with respect to the given quantitative objective. FamilySketcher and Brute-Force use the polyhedra domain as parameter.

The Loop1a and Loop1b sketches are handled symbolically by (SR) rule. Thus, our approach does not depend on sizes of hole domains. FamilySketcher terminates in (around) 0.016 sec for Loop1a and in 0.026 sec for Loop1b. In contrast, Brute-Force and Sketch do depend on the sizes of holes. Sketch terminates in 37.74 sec (resp., 2.44 sec) for 16-bits sizes of holes for Loop1a (resp., Loop1b). It times out for bigger sizes of Loop1a. Sketch often reports "correct & optimal" solutions for both sketches. Similarly, our tool can handle symbolically Loop2a and Loop2b in 0.060 sec and 0.047 sec. However, Sketch cannot resolve them, since it uses 8 unrollments of the loop by default. If the loop is unrolled 11 times, Sketch terminates but often reports the empty solution.

The LoopCond sketch contains one hole that can be handled symbolically by (SR) rule. FamilySketcher has similar running times for all domain sizes reporting the solution $\texttt{??} \geq 2$ and ranking function 4x+8. In contrast, Sketch 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.

The NestedLoop sketch contains two holes that can be handled symbolically by (SR) rule. FamilySketcher terminates in (around) 0.126 sec for all sizes of holes. The "correct" solution is $(\texttt{??}_1 - \texttt{??}_2 \geq 0) \wedge (\texttt{Min} \leq \texttt{??}_2 \leq 1)$, while the "correct & optimal" solution is $(\texttt{??}_1 = \texttt{??}_2 = 0)$ with ranking function 13. On the other hand, Brute-Force takes 65.03 sec for 5-bit size of holes and times out for larger sizes, while Sketch cannot resolve this benchmark.

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 $\texttt{??}_2$. 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 $??_2$). The sub-family of "correct" solutions is: $(1 \leq \texttt{A} \leq \texttt{Max}) \wedge (\texttt{B} \geq 10)$, while the "correct & optimal" solution is $(\texttt{A}{=}1) \wedge (\texttt{B}{=}10)$ with ranking function 6. Sketch scales better in this case reporting one "correct" (but not "optimal") solution. However, FamilySketcher still outperforms the Brute-Force approach.

Table 1: Performance results of FamilySketcher vs. `Sketch` vs. `Brute-Force`. FamilySketcher and `Brute-Force` use Polyhedra domain. All times in sec.

Bench.	5 bits			6 bits			16 bits		
	FamilySketcher	Sketch	Brute Force	FamilySketcher	Sketch	Brute Force	FamilySketcher	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
LoopCond	0.042	0.207	1.19	0.042	0.209	2.56	0.043	timeout	timeout
NestedLoop	0.126	timeout	65.03	0.126	timeout	timeout	0.128	timeout	timeout
vmcai2004	4.69	0.192	5.12	15.52	0.229	19.12	timeout	0.292	timeout

Discussion In summary, we can see that FamilySketcher often outperforms `Sketch`, especially in case of sketches that can be handled symbolically by (SR) rule. But, for 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 values of those holes. However, even in this case FamilySketcher scales better than `Brute-Force`. 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, while `Sketch` reports the first found "correct" solution.

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. We perform lifted analysis of relatively small benchmarks. However, the focus of our approach is to combat the realization space blow-up of sketches, not their LOC size. So, we expect to obtain similar or better results for larger benchmarks. Although we analyze relatively small set of benchmarks, we expect the results to carry over the other benchmarks.

7 Related Work

The proposed program sketcher uses numerical abstract domains as parameters, so it can be applied for synthesizing programs with numerical data types. The existing widely-known sketching tool `Sketch` [29,30], 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` 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, "correct" solutions. This is the key for applying our approach to solve the quantitative sketching problem. 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 [5]. 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 worst-case behavior of the program is minimal.

Recently, there have been proposed several works that solve the sketching synthesis problem using product line analysis and verification algorithms. Ceska et. al. [4] use a counterexample guided abstraction refinement technique for analyzing product lines to resolve probabilistic `PRISM` sketches. The work [17] uses a (forward) numerical lifted analysis based on abstract interpretation to resolve numerical sketches. We extend here this approach by considering the more general quantitative sketching problem, where we additionally employ a (backward) termination lifted analysis to find a solution that is not only "correct" but also "optimal" to the given quantitative objective.

Several lifted analysis based on abstract interpretation have been proposed recently [24,11,12,16,18,15,13] for analyzing program families with `#if`-s. Midtgaard et. al. [24] have proposed the lifted tuple-based analysis, while the work [11,12] 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 [16,18], as well as dynamic program families where features can change during run-time [15]. The above lifted analyses are forward and infer numerical invariants, while a backward termination analysis for inferring ranking functions is proposed in [13].

Decision-tree abstract domains have been used in abstract interpretation community recently [10,7,32]. Segmented decision tree abstract domains have enabled path dependent static analysis [10,7]. Their elements contain decision nodes that are determined either by values of program variables [10] or by the `if` conditions [7], whereas the leaf nodes are numerical properties. Urban and Miné [31,32] use decision tree abstract domains to prove program termination.

8 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. We experimentally demonstrate the effectiveness of our approach on a variety of C benchmarks.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013. pp. 1–8. IEEE (2013), http://ieeexplore.ieee.org/document/6679385/
2. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings. LNCS, vol. 5643, pp. 140–156. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_14, https://doi.org/10.1007/978-3-642-02658-4_14
3. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years. In: ACM SIGPLAN Conference on PLDI '13. pp. 355–364 (2013)
4. Ceska, M., Dehnert, C., Jansen, N., Junges, S., Katoen, J.: Model repair revamped: On the automated synthesis of markov chains. In: Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday. LNCS, vol. 11500, pp. 107–125. Springer (2019). https://doi.org/10.1007/978-3-030-31514-6_7, https://doi.org/10.1007/978-3-030-31514-6_7
5. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Computer Aided Verification, 22nd International Conference, CAV 2010. Proceedings. LNCS, vol. 6174, pp. 380–395. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_34, https://doi.org/10.1007/978-3-642-14295-6_34
6. Chaudhuri, S., Clochard, M., Solar-Lezama, A.: Bridging boolean and quantitative synthesis using smoothed proof search. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14. pp. 207–220. ACM (2014). https://doi.org/10.1145/2535838.2535859, https://doi.org/10.1145/2535838.2535859
7. Chen, J., Cousot, P.: A binary decision tree abstract domain functor. In: Static Analysis - 22nd International Symposium, SAS 2015, Proceedings. LNCS, vol. 9291, pp. 36–53. Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_3, https://doi.org/10.1007/978-3-662-48288-9_3
8. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
9. Cousot, P., Cousot, R.: 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. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, http://doi.acm.org/10.1145/512950.512973
10. Cousot, P., Cousot, R., Mauborgne, L.: A scalable segmented decision tree abstract domain. In: Time for Verification, Essays in Memory of Amir Pnueli. LNCS, vol. 6200, pp. 72–95. Springer (2010). https://doi.org/10.1007/978-3-642-13754-9_5, https://doi.org/10.1007/978-3-642-13754-9_5
11. Dimovski, A.S.: Lifted static analysis using a binary decision diagram abstract domain. In: Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019. pp. 102–114. ACM (2019). https://doi.org/10.1145/3357765.3359518, https://doi.org/10.1145/3357765.3359518
12. Dimovski, A.S.: A binary decision diagram lifted domain for analyzing program families. J. Comput. Lang. **63**, 101032 (2021).

https://doi.org/10.1016/j.cola.2021.101032, https://doi.org/10.1016/j.cola.2021.101032
13. Dimovski, A.S.: Lifted termination analysis by abstract interpretation and its applications. In: GPCE '21: Concepts and Experiences, Chicago, IL, USA, October, 2021. pp. 96–109. ACM (2021). https://doi.org/10.1145/3486609.3487202, https://doi.org/10.1145/3486609.3487202
14. Dimovski, A.S.: Tool artifact for "quantitative program sketching using lifted static analysis". Zenodo (2022). https://doi.org/10.5281/zenodo.5898643, https://zenodo.org/record/5898643#.YhJLRejMLIU
15. Dimovski, A.S., Apel, S.: Lifted static analysis of dynamic program families by abstract interpretation. In: 35th European Conference on Object-Oriented Programming, ECOOP 2021. LIPIcs, vol. 194, pp. 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.14, https://doi.org/10.4230/LIPIcs.ECOOP.2021.14
16. Dimovski, A.S., Apel, S., Legay, A.: A decision tree lifted domain for analyzing program families with numerical features. In: Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings. LNCS, vol. 12649, pp. 67–86. Springer (2021), https://arxiv.org/abs/2012.05863
17. Dimovski, A.S., Apel, S., Legay, A.: Program sketching using lifted analysis for numerical program families. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings. LNCS, vol. 12673, pp. 95–112. Springer (2021). https://doi.org/10.1007/978-3-030-76384-8_7, https://doi.org/10.1007/978-3-030-76384-8_7
18. Dimovski, A.S., Apel, S., Legay, A.: Several lifted abstract domains for static analysis of numerical program families. Sci. Comput. Program. **213**, 102725 (2022). https://doi.org/10.1016/j.scico.2021.102725, https://doi.org/10.1016/j.scico.2021.102725
19. Dimovski, A.S., Brabrand, C., Wasowski, A.: Variability abstractions for lifted analysis. Sci. Comput. Program. **159**, 1–27 (2018)
20. Dimovski, A.S., Brabrand, C., Wasowski, A.: Finding suitable variability abstractions for lifted analysis. Formal Aspects Comput. **31**(2), 231–259 (2019). https://doi.org/10.1007/s00165-019-00479-y, https://doi.org/10.1007/s00165-019-00479-y
21. Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., Apel, S.: Preprocessor-based variability in open-source and industrial software systems: An empirical study. Empirical Software Engineering **21**(2), 449–482 (2016). https://doi.org/10.1007/s10664-015-9360-1, https://doi.org/10.1007/s10664-015-9360-1
22. Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective analysis of C programs by rewriting variability. Art Sci. Eng. Program. **1**(1), 1 (2017). https://doi.org/10.22152/programming-journal.org/2017/1/1, https://doi.org/10.22152/programming-journal.org/2017/1/1
23. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings. LNCS, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52, https://doi.org/10.1007/978-3-642-02658-4_52
24. Midtgaard, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Systematic derivation of correct variability-aware program analyses. Sci. Comput. Program. **105**, 145–170 (2015). https://doi.org/10.1016/j.scico.2015.04.005, http://dx.doi.org/10.1016/j.scico.2015.04.005

25. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages **4**(3-4), 120–372 (2017). https://doi.org/10.1561/2500000034, https://doi.org/10.1561/2500000034
26. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
27. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag, Secaucus, USA (1999)
28. von Rhein, A., Liebig, J., Janker, A., Kästner, C., Apel, S.: Variability-aware static analysis at scale: An empirical study. ACM Trans. Softw. Eng. Methodol. **27**(4), 18:1–18:33 (2018). https://doi.org/10.1145/3280986, https://doi.org/10.1145/3280986
29. Solar-Lezama, A.: Program sketching. STTT **15**(5-6), 475–495 (2013). https://doi.org/10.1007/s10009-012-0249-7, https://doi.org/10.1007/s10009-012-0249-7
30. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. pp. 281–294. ACM (2005). https://doi.org/10.1145/1065010.1065045, https://doi.org/10.1145/1065010.1065045
31. Urban, C.: 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), https://tel.archives-ouvertes.fr/tel-01176641
32. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Static Analysis - 21st International Symposium, SAS 2014. Proceedings. LNCS, vol. 8723, pp. 302–318. Springer (2014). https://doi.org/10.1007/978-3-319-10936-7_19, https://doi.org/10.1007/978-3-319-10936-7_19