Program Sketching using Lifted Analysis for Numerical Program Families

Aleksandar S. Dimovski ✉ [1], Sven Apel [2], and Axel Legay [3]

[1] Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia
aleksandar.dimovski@unt.edu.mk
[2] Saarland University, Saarland Informatics Campus, E1.1, 66123 Saarbrücken, Germany
[3] Université catholique de Louvain, 1348 Ottignies-Louvain-la-Neuve, Belgium

Abstract. This work presents a novel approach for synthesizing numerical *program sketches* using *lifted* (*family-based*) *static program analysis*. In particular, our approach leverages a lifted static analysis based on abstract interpretation, which is used for analyzing program families with numerical features. It takes as input the common code base, which encodes all variants of a program family, and produces precise results for all variants in a single analysis run. The elements of the underlying lifted analysis domain are *decision trees*, in which decision nodes are labeled with linear constraints defined over numerical features and leaf nodes belong to a given single-program analysis domain.
We encode a program sketch as a program family such that holes correspond to numerical features and all possible sketch realizations correspond to variants in the program family. Then, we preform a lifted analysis of the family, so that only those variants that satisfy all assertions under all possible inputs represent correct realizations of holes in the sketch.
We have implemented an experimental program synthesizer for resolving C sketches. It is based on a lifted static analyzer for `#if`-based C program families, which uses the numerical domains from the APRON library. An evaluation yields promising results. Moreover, our approach provides speedups in some cases against the popular sketching tool `Sketch` and can solve some numerical benchmarks that `Sketch` cannot handle.

1 Introduction

Sketching [23,24] is one of the earliest and successful forms of program synthesis [1]. A *sketch* is a partial program that expresses the high-level structure of an implementation but leaves *holes* in place of low-level details. An integer hole is a placeholder that the synthesizer must replace with one of finitely many constant values. More specifically, the user provides the specification of the required program in the form of assertions, as well as a partial program with holes that needs to be completed. The goal of synthesizer is to automatically find integer values for the holes so that the resulting complete program satisfies the assertions under all possible inputs. To solve the sketching problem, we need algorithms that can efficiently search to fill in holes without any user support.

In this paper, we rely on the notion of program families (a.k.a. software product lines) [2,7] with numerical features to formalize this problem. We apply specifically designed lifted static analysis algorithms, which operate directly on program families rather than on individual programs, to solve the sketching problem. A *program family* is a set of similar, tailor-made programs, called *variants*, that is built from a common code base [21]. The variants of a program family are specified in terms of *features* (configuration options) that are statically selected for that particular variant at compile-time. In particular, we consider program families implemented using `#if` directives from the C preprocessor `CPP` [7,17]. An `#if` directive specifies under which conditions parts of code should be included or excluded from a variant. At compile-time, a variant is derived by assigning concrete values to features, and only then is this variant compiled.

A key idea is that all possible program sketch realizations constitute a program family, where each integer hole is represented as a numerical feature. This way, program sketching is all about selecting correct variants (family members) from the corresponding program family. However, the automated analysis of program families for finding a correct variant is challenging since the family size (i.e., number of variants) typically grows exponentially in the number of features (i.e., holes). This is particulary apparent in the case of program families that contain numerical features with big domains, thus admitting astronomic family sizes (configuration spaces). Program sketching is also affected by this problem, since the family size corresponds to the space of sketch realizations. A naive enumerative (brute-force) solution, which analyzes each individual variant of the program family by an existing off-the-shelf single-program analyzer, has been shown to be very inefficient for large families [3,20].

To address the program sketching problem, we use a *lifted (a.k.a. family-based or variability-aware) static program analysis* [3,28,20,11,13,18,12], which analyzes all variants of the family simultaneously, without generating any of them explicitly. Lifted analysis processes the common code base of a program family directly, exploiting the similarities among individual variants to reduce analysis effort. It reports analysis results for all variants that are equivalent to what a brute force approach would report. In particular, here we use an efficient, abstract interpretation-based lifted analysis of program families with numerical features [13], where *sharing* is explicitly possible between analysis elements corresponding to different variants. Inspired by the decision tree abstract domain proposed by Urban and Mine [27,26,25] for proving program termination, this is achieved by defining a specialized *decision tree lifted domain* [13] 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 numerical features, while leaf nodes belong to a given single-program analysis domain. The decision trees recursively partition the space of variants (i.e., the space of possible combinations of feature 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 lifted analysis

```
void  main(unsigned int x){
①         int y;
②         y := x*??;
③         assert (y ≤ x+x); ④ }
```

Fig. 1: HelloWorld sketch.

```
void  main() {
①     int x := ??, y := 0;
②     while (x > ??) {
③         x := x-1;
④         y := y+1; }
⑤     assert (y > 2); ⑥ }
```

Fig. 2: Loop sketch.

partitions the given family into: "good" (correct), "bad" (incorrect), and "I don't know" (inconclusive) variants, i.e., sketch completions, with respect to a given set of assertions. Because of its special structure and possibilities for sharing of equivalent analysis results, the decision tree-based lifted analysis is able to converge to a solution very fast even for program families with astronomical search spaces. This is particularly so for sketches in which holes appear in expressions that can be exactly represented in the underlying domains for decision and leaf nodes (e.g., polyhedra). In those cases, we can design efficient lifted analysis with extended transfer functions for assignments and tests. Our approach is *sound* but *incomplete*: whatever correct sketch completions are inferred they can be trusted to hold, but we can miss some correct sketch completions.

Contributions. In summary, we make several contributions: (1) We propose a new, efficient technique for solving numerical program sketches by using lifted static analysis; (2) We implement a prototype program sketcher, called FamilySketcher, that uses numerical domains (e.g., polyhedra) from the APRON library as parameters; (3) We evaluate our approach on several C numerical sketches and compare its performances with the popular sketching tool `Sketch` and the brute-force enumeration approach.

Motivating examples. The code snippet HelloWorld in Fig. 1 is regarded as the "Hello World" example of sketching [23]. This sketch contains one integer hole, denoted by `??`. Note that `x` is an input variable that can take non-negative integer values. As observed before in the literature [5], a sketch can be represented as a program family, such that all possible realizations of holes in the sketch correspond to possible variants in the program family. The HelloWorld sketch can be encoded as a program family that contains one numerical feature `A` with domain $[\mathtt{Min}, \mathtt{Max}] \subseteq \mathbb{Z}$, such that the hole `??` is replaced with `A` [4]. There are $\mathtt{Max} - \mathtt{Min} + 1$ variants that can be derived from the HelloWorld program family. To find a correct variant that satisfies the given assertion, we perform a lifted analysis based on decision trees [13] of the corresponding HelloWorld program family. The decision tree (invariant) inferred at the final location ③ when `A` has domain $[0, 3]$ is shown in Fig. 3. Notice that the inner nodes of the decision tree in Fig. 3 are labeled with *polyhedral* linear constraints over the numerical feature `A`, while the leaves are labeled with *polyhedral* linear constraints over program

[4] This is only high-level description of the encoding. For the precise definition, we refer to Section 3.3. See the HelloWorld program family in Fig. 5a

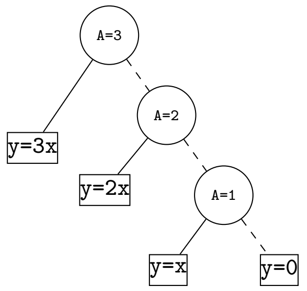

Fig. 3: Tree at loc. ③ of HelloWorld.

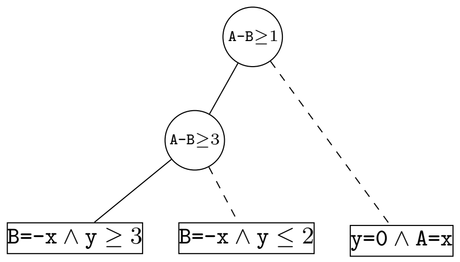

Fig. 4: Tree at loc. ⑤ of Loop.

variables x and y. 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). As decision nodes partition the space of all possible feature's values, we implicitly assume that linear constraints in decision nodes take domains of numerical features into account. For example, the decision node $(\mathtt{A}=3)$ is satisfied when $(\mathtt{A}=3) \land (\mathtt{Min} \leq \mathtt{A} \leq \mathtt{Max})$, whereas its negation is satisfied when $(\mathtt{A} \neq 3) \land (\mathtt{Min} \leq \mathtt{A} \leq \mathtt{Max})$. The final assertion is valid at leaf nodes when $(\mathtt{A} \leq 2)$. Hence, we can replace the hole ?? in the sketch with an integer less or equal than 2, so that the assertion is always valid.

Consider the Loop sketch in Fig. 2 taken from Syntax-Guided Synthesis Competition (https://sygus.org/) [1]. It contains two integer holes, which are replaced with two numerical features A and B in the Loop program family. Since the domain of each numerical feature is $[\mathtt{Min}, \mathtt{Max}]$, the total number of variants is $(\mathtt{Max} - \mathtt{Min} + 1)^2$. If we analyze the Loop program family using a lifted analysis based on decision trees as before, we obtain the invariant shown in Fig. 4 at the location ⑤. We can see that the given assertion is valid for $(\mathtt{A\text{-}B} \geq 3)$. Thus, the synthesizer can choose one of the variants that satisfy the above constraint (e.g., A=4, B=1) as a solution to the Loop sketch.

2 From Sketches to Program Families

Now we introduce the languages for writing sketches and program families. Then, we define the encoding of sketches as program families and show its correctness.

2.1 Sketches

We use a simple sequential imperative language to illustrate our work. The program variables *Var* are statically allocated, and the only data type is the set $\mathbb{Z}$ of mathematical integers. To encode sketches, a single sketching construct is included: a basic integer hole denoted by ??. The integer hole ?? is a placeholder that the synthesizer must replace with a suitable integer constant, such that the resulting program will avoid any assertion failures. The syntax is:

$$s ::= \mathtt{skip} \mid \mathtt{x}\!:=\!e \mid s;s \mid \mathtt{if}\ (e)\ \mathtt{then}\ s\ \mathtt{else}\ s \mid \mathtt{while}\ (e)\ \mathtt{do}\ s \mid \mathtt{assert}(e)$$
$$e ::= n \mid \mathtt{x} \mid \mathtt{??} \mid e \oplus e$$

where n ranges over integers, $\mathtt{x}$ over program variables Var, and $\oplus$ over binary arithmetic-logic operators. Each hole occurrence is assumed to be uniquely labelled as $\mathtt{??}_i$ and has a bounded integer domain $[n, n']$. We will sometimes write $\mathtt{??}_i^{[n,n']}$ to make explicit the domain of a given hole. Our aim is to replace each $\mathtt{??}_i^{[n,n']}$ with a suitable constant from $[n, n']$. By *Stm* and *Exp* we denote the set of statements s and expressions e generated by the above grammar. W.l.o.g., a program is a sequence of statements (without `assert`) followed by an assertion.

Semantics. 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. A *program state* $\sigma : \Sigma = \mathit{Var} \to \mathbb{Z}$ is a mapping from variables to values. The meaning of expressions $[\![e]\!] : \Sigma \to \Phi \to \mathbb{Z}$ is defined by induction on e:

$$[\![n]\!]\sigma\phi = n,\ [\![\mathtt{x}]\!]\sigma\phi = \sigma(\mathtt{x}),\ [\![\mathtt{??}_i]\!]\sigma\phi = \phi(\mathtt{??}_i),\ [\![e_0 \oplus e_1]\!]\sigma\phi = [\![e_0]\!]\sigma\phi \oplus [\![e_1]\!]\sigma\phi$$

The inference rules for a small-step operational semantics of statements are standard [20]. We write $[\![s]\!]\sigma\phi$ for the final state σ' that can be derived from $\langle s, \sigma\phi\rangle$ (if the derivation terminates successfully), that is $\langle s, \sigma\phi\rangle \to^* \sigma'$, by using the inference rules. The meaning of statements is: $[\![s]\!]^\phi = \bigcup_{\sigma\in\Sigma^{\mathtt{init}}} [\![s]\!]\sigma\phi$, where $\Sigma^{\mathtt{init}}$ denotes the set of initial input states on which s is executed.

2.2 Program Families

Let $\mathbb{F} = \{A_1, \ldots, A_k\}$ be a finite and totaly ordered set of *numerical features* available in a program family. For each feature $A \in \mathbb{F}$, $\mathrm{dom}(A) \subseteq \mathbb{Z}$ denotes the set of possible values that can be assigned to A. A valid combination of feature values represents a *configuration* k, which specifies one *variant* of a program family. It is given as a *valuation function* $k : \mathbb{F} \to \mathbb{Z}$, which is a mapping that assigns a value from $\mathrm{dom}(A)$ to each feature A, that is, $k(A) \in \mathrm{dom}(A)$ for any $A \in \mathbb{F}$. 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: $(A_1 = k(A_1)) \land \ldots \land (A_k = k(A_k))$. The set of valid configurations $\mathbb{K}$ can be also represented as a formula: $\lor_{k\in\mathbb{K}} k$.

We define *feature expressions*, denoted $\mathit{FeatExp}(\mathbb{F})$, as the set of propositional logic formulas over constraints of $\mathbb{F}$ generated by the grammar:

$$\theta ::= \mathrm{true} \,|\, e_{\mathbb{F}} \bowtie e_{\mathbb{F}} \,|\, \neg\theta \,|\, \theta \land \theta \,|\, \theta \lor \theta, \qquad e_{\mathbb{F}} ::= n \mid A \mid e_{\mathbb{F}} \boxplus e_{\mathbb{F}}$$

where $A \in \mathbb{F}$, $n \in \mathbb{Z}$, $\boxplus \in \{+, -, *\}$, and $\bowtie \in \{=, <\}$. When a configuration $k \in \mathbb{K}$ satisfies a feature expression $\theta \in \mathit{FeatExp}(\mathbb{F})$, we write $k \models \theta$, where $\models$ is the standard satisfaction relation of logic. We write $[\![\theta]\!]$ to denote the set of configurations from $\mathbb{K}$ that satisfy θ, that is, $k \in [\![\theta]\!]$ iff $k \models \theta$. For example, for the HelloWorld program family in Fig. 5a we have $\mathbb{F} = \{\mathtt{A}\}$ and $\mathbb{K} = \{(\mathtt{A} = \mathtt{Min}), \ldots, (\mathtt{A} = \mathtt{Max})\}$. For the feature expression $(\mathtt{A} > 2)$, we have $[\![(\mathtt{A}>2)]\!] = \{(\mathtt{A}=3), \ldots (\mathtt{A}=\mathtt{Max})\}$. Thus, $(\mathtt{A}=5) \models (\mathtt{A}>2)$ and $(\mathtt{A}=0) \not\models (\mathtt{A}>2)$.

```
int y;
#if (A=Min) y := x*Min
 #elif ...
 #elif (A=Max-1) y := x*(Max-1)
 #else y := x*Max ... #endif
assert (y ≤ x+x);
```

(a) HelloWorld

```
int y;
y := x*0;
assert(y ≤ x+x);
```

(b) $P_{A=0}$(HelloW.)

```
int y;
y := x*1;
assert(y ≤ x+x);
```

(c) $P_{A=1}$(HelloW.)

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

(d) Loop

Fig. 5: The program families HelloWorld and Loop and some their variants.

The language includes the same expression and statement productions as the language for sketches, except that the hole expression `??` is not allowed. To encode multiple variants, a new compile-time conditional statement is included. The new statement "`#if` (θ) $\overline{s}$ `#endif`" contains a feature expression $\theta \in \mathit{FeatExp}(\mathbb{F})$ as a *presence condition*, such that only if θ is satisfied by a configuration $k \in \mathbb{K}$ the statement $\overline{s}$ will be included in the variant corresponding to k. The syntax is:

$$\overline{s} ::= \ldots \mid \texttt{\#if}\ (\theta)\ \overline{s}\ \texttt{\#endif}, \qquad \overline{e} ::= n \mid \texttt{x} \mid \overline{e} \oplus \overline{e}$$

The set of all statements $\overline{s}$ is $\overline{Stm}$; the set of all expressions $\overline{e}$ is $\overline{Exp}$. Any other preprocessor conditional constructs can be desugared and represented only by `#if` construct. For example, `#if` (θ) $\overline{s_0}$ `#elif` (θ') $\overline{s_1}$ `#endif` is translated into `#if` (θ) $\overline{s_0}$ `#endif ; #if` $(\neg\theta \wedge \theta')$ $\overline{s_1}$ `#endif`.

Semantics. A program family is evaluated in two stages. First, the preprocessor `CPP` takes a program family $\overline{s}$ and a configuration $k \in \mathbb{K}$ as inputs, and produces a variant (that is, a single program without `#if`-s) corresponding to k as the output. Second, the obtained variant is evaluated using the standard single-program semantics [20]. The first stage is specified by the projection function π_k, which is an identity for all basic statements and recursively pre-processes all sub-statements of compound statements. Hence, $\pi_k(\texttt{skip}) = \texttt{skip}$ and $\pi_k(\overline{s};\overline{s'}) = \pi_k(\overline{s});\pi_k(\overline{s'})$. For "`#if` (θ) $\overline{s}$ `#endif`", statement $\overline{s}$ is included in the variant if $k \models \theta$, otherwise, if $k \not\models \theta$ statement $\overline{s}$ is removed [5]: $\pi_k(\texttt{\#if}\ (\theta)\ \overline{s}\ \texttt{\#endif}) = \begin{cases} \pi_k(\overline{s}) & \text{if } k \models \theta \\ \texttt{skip} & \text{if } k \not\models \theta \end{cases}$. For example, Figure 5a shows the code of the program family HelloWorld (only the function body) that contains one numerical feature `A` with domain $[\texttt{Min}, \texttt{Max}]$. Two valid variants $\texttt{P}_{\texttt{A}=0}$(HelloWorld) and $\texttt{P}_{\texttt{A}=1}$(HelloWorld) shown in Fig. 5b and Fig. 5c, respectively, can be derived from the HelloWorld family in Fig. 5a.

We define the semantics of variants $\pi_k(\overline{s})$, i.e. single programs without `#if`-s. A *program state* is $\sigma : \Sigma = \mathit{Var} \to \mathbb{Z}$. The meaning of expressions $[\![\overline{e}]\!] : \Sigma \to \mathbb{Z}$ is:

$$[\![n]\!]\sigma = n, \qquad [\![\texttt{x}]\!]\sigma = \sigma(\texttt{x}), \qquad [\![\overline{e_1} \oplus \overline{e_2}]\!]\sigma = [\![\overline{e_1}]\!]\sigma \oplus [\![\overline{e_2}]\!]\sigma$$

The set of statements is the same for sketches and variants, so the meaning of statements $[\![\pi_k(\overline{s})]\!]$ for variants coincides with the meaning for sketches.

[5] 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 $\theta \in \mathit{FeatExp}(\mathbb{F})$.

2.3 Encoding of Sketches as Program Families

Our aim is to transform an input program sketch $\hat{s}$ with a set of m holes $\texttt{??}_1^{[n_1,n'_1]}, \dots, \texttt{??}_m^{[n_m,n'_m]}$ into an output program family $\overline{s}$ with a set of numerical features $A_1, \dots, A_m$ with domains $[n_1, n'_1], \dots, [n_m, n'_m]$, respectively. The set of configurations $\mathbb{K}$ includes all possible combinations of feature values.

We now define a rewrite rule for eliminating holes $\texttt{??}$ from a program sketch. Let $s[\texttt{??}^{[n,n']}]$ be a basic (non-compound) statement in which the hole $\texttt{??}^{[n,n']}$ occurs. The rewrite rule is of the form:

$$s[\texttt{??}^{[n,n']}] \;\rightsquigarrow\; \texttt{\#if (A=}n\texttt{)}\; s[n]\; \texttt{\#elif} \dots \texttt{\#elif (A=}n'\texttt{-1)}\; s[n'\texttt{-1}] \quad \texttt{\#else}\; s[n']\; \texttt{\#endif} \dots \texttt{\#endif} \tag{R-1}$$

The meaning of rule (R-1) is that, if the current program sketch being transformed matches the abstract syntax tree node of the shape $s[\texttt{??}^{[n,n']}]$, then replace $s[\texttt{??}^{[n,n']}]$ with the right-hand side of rule (R-1). The set of features $\mathbb{F}$ is also updated with the fresh feature $\texttt{A}$ with domain $[n, n']$.

We write $\texttt{Rewrite}(\hat{s})$ to be the final transformed program family $\overline{s}$ obtained by repeatedly applying rule (R-1) on a program sketch $\hat{s}$ and on its transformed versions until we reach a point at which this rule can not be applied, i.e. when all occurrences of holes $\texttt{??}$ in $\hat{s}$ are eliminated. For example, the program sketch HelloWorld in Fig. 1 is encoded as the program family HelloWorld in Fig. 5a.

Theorem 1. *Let $\hat{s}$ be a sketch and ϕ be a control function, s.t. features $A_1, \dots, A_n$ correspond to holes $\texttt{??}_1, \dots, \texttt{??}_n$. We define a configuration $k \in \mathbb{K}$, s.t. $k(A_i) = \phi(\texttt{??}_i)$ for $1 \le i \le n$. Let $\overline{s} = \texttt{Rewrite}(\hat{s})$. We have: $[\![\hat{s}]\!]^{\phi} = [\![\pi_k(\overline{s})]\!]$.*

3 Synthesis by Lifted Analysis

Lifted analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. Lifted analysis by abstract interpretation introduced in [13] relies on a lifted domain in the form of *decision trees* [27,26]. The leaf nodes of decision trees belong to an existing single-program analysis domain, and are separated by linear constraints over numerical features, organized in decision nodes. We first define the basic decision tree lifted domain in Section 3.1. Then in Section 3.2, we show how we can optimize the encoding of sketches as program families and extend the decision tree lifted domain to obtain more efficient program sketcher. Finally, in Section 3.3, we present a synthesis algorithm for resolving sketches based on lifted analysis.

3.1 Basic Lifted Analysis

Abstract domain for leaf nodes. We assume that a single-program domain $\mathbb{A}$ defined over program variables *Var* 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 FILTER$_{\mathbb{A}}$

and forward assignments $\text{ASSIGN}_{\mathbb{A}}$. More specifically, $\text{FILTER}_{\mathbb{A}}(a : \mathbb{A}, e : \mathit{Exp})$ returns an abstract element from $\mathbb{A}$ obtained by restricting a to satisfy the test e, whereas $\text{ASSIGN}_{\mathbb{A}}(a : \mathbb{A}, \texttt{x:=}e : \mathit{Stm})$ returns an updated version of a by abstractly evaluating $\texttt{x:=}e$ in it. In practice, the domain $\mathbb{A}$ will be instantiated with some of the known numerical domains $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$, such as Intervals $\langle I, \sqsubseteq_I \rangle$ [8], Octagons $\langle O, \sqsubseteq_O \rangle$ [27], and Polyhedra $\langle P, \sqsubseteq_P \rangle$ [10], defined over Var. For example, 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 $x_1, \ldots x_k \in \mathit{Var}, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}$.

Abstract domain for decision nodes. We introduce a family of abstract domains for linear constraints $\mathbb{C}_{\mathbb{D}}$ defined over features $\mathbb{F}$, which are parameterized by any of the numerical property domains $\mathbb{D}$ (intervals I, octagons O, polyhedra P). For example, the set of *polyhedral constraints* is $\mathbb{C}_P = \{\alpha_1 A_1 + \ldots + \alpha_k A_k + \beta \geq 0 \mid A_1, \ldots A_k \in \mathbb{F}, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}, \gcd(|\alpha_1|, \ldots, |\alpha_k|, |\beta|) = 1\}$. The set $\mathbb{C}_{\mathbb{D}}$ of linear constraints over features $\mathbb{F}$ is constructed by the underlying numerical property domain $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$ using the Galois connection $\langle \mathcal{P}(\mathbb{C}_{\mathbb{D}}), \sqsubseteq_{\mathbb{D}} \rangle \underset{\alpha_{\mathbb{C}_{\mathbb{D}}}}{\overset{\gamma_{\mathbb{C}_{\mathbb{D}}}}{\leftrightarrows}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$, where $\mathcal{P}(\mathbb{C}_{\mathbb{D}})$ is the power set of $\mathbb{C}_{\mathbb{D}}$ [13]. The concretization function $\gamma_{\mathbb{C}_{\mathbb{D}}} : \mathbb{D} \to \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ maps a conjunction of constraints from $\mathbb{D}$ to a set of constraints in $\mathcal{P}(\mathbb{C}_{\mathbb{D}})$.

The domain of decision nodes is $\mathbb{C}_{\mathbb{D}}$. We assume the set of features $\mathbb{F} = \{A_1, \ldots, A_k\}$ to be totally ordered, such that the ordering is $A_1 > \ldots > A_k$. 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:

$$\begin{aligned}&(\alpha_1 \cdot A_1 + \ldots + \alpha_k \cdot A_k + \alpha_{k+1} \geq 0) \;<_{\mathbb{C}_{\mathbb{D}}}\; (\alpha'_1 \cdot A_1 + \ldots + \alpha'_k \cdot A_k + \alpha'_{k+1} \geq 0)\\ &\iff \exists j > 0. \forall i < j. (\alpha_i = \alpha'_i) \wedge (\alpha_j < \alpha'_j)\end{aligned}$$

The negation of linear constraints is formed as: $\neg(\alpha_1 A_1 + \ldots \alpha_k A_k + \beta \geq 0) = -\alpha_1 A_1 - \ldots - \alpha_k A_k - \beta - 1 \geq 0$. For example, the negation of $A - 3 \geq 0$ is $-A + 2 \geq 0$. 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 $\neg c$.

Abstract domain for decision trees. A *decision tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ over the sets $\mathbb{C}_{\mathbb{D}}$ of linear constraints defined over $\mathbb{F}$ and the leaf abstract domain $\mathbb{A}$ defined over Var is either a leaf node $\ll a \gg$ with $a \in \mathbb{A}$, or $[\![c : tl, tr]\!]$, where $c \in \mathbb{C}_{\mathbb{D}}$ (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 the analysis properties for the corresponding configurations.

Example 1. The following two decision trees t_1 and t_2 have decision and leaf nodes labelled with polyhedral linear constraints defined over numerical feature $\texttt{A}$ with domain $\mathbb{Z}$ and over integer program variable y, respectively:

$$t_1 = [\![\texttt{A} \geq 4 : \ll [y \geq 2] \gg, \ll [y = 0] \gg]\!], \; t_2 = [\![\texttt{A} \geq 2 : \ll [y \geq 0] \gg, \ll [y \leq 0] \gg]\!] \qquad \square$$

Algorithm 1: $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}e)$

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

Abstract 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 $k \in \mathbb{K}$, where k satisfies the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ of constraints accumulated along the top-down path to the leaf node $a \in \mathbb{A}$. More formally, $\gamma_{\mathbb{T}}(t) = \overline{\gamma}_{\mathbb{T}}[\mathbb{K}](t)$, where $\mathbb{K} = \vee_{k\in\mathbb{K}} k$ is the set of implicit constraints over $\mathbb{F}$ taking into account the domains of features. Function $\overline{\gamma}_{\mathbb{T}}$ is defined as:

$$\overline{\gamma}_{\mathbb{T}}[C](\ll a\gg) = \textstyle\prod_{k\models C}\gamma_{\mathbb{A}}(a), \quad \overline{\gamma}_{\mathbb{T}}[C]([\![c : tl, tr]\!]) = \overline{\gamma}_{\mathbb{T}}[C\cup\{c\}](tl) \times \overline{\gamma}_{\mathbb{T}}[C\cup\{\neg c\}](tr)$$

Other binary operations rely on the algorithm for *tree unification* [13,26], which finds a common labelling of two trees t_1 and t_2. Note that the tree unification does not lose any information.

Example 2. After tree unification of t_1 and t_2 from Example 1, we obtain:

$$\begin{aligned} t_1 &= [\![\texttt{A} \geq 4 : \ll[y \geq 2]\gg, [\![\texttt{A} \geq 2 : \ll[y = 0]\gg, \ll[y = 0]\gg]\!]]\!], \\ t_2 &= [\![\texttt{A} \geq 4 : \ll[y \geq 0]\gg, [\![\texttt{A} \geq 2 : \ll[y \geq 0]\gg, \ll[y \leq 0]\gg]\!]]\!] \end{aligned}$$

Note that the tree unification adds a decision node for $\texttt{A} \geq 2$ to the right subtree of t_1, whereas it adds a decision node for $\texttt{A} \geq 4$ to t_2 and removes the redundant constraint $\texttt{A} \geq 2$ from the resulting left subtree of t_2. □

All binary operations are performed leaf-wise on the unified decision trees. Given two unified trees t_1 and t_2, their ordering $t_1 \sqsubseteq_{\mathbb{T}} t_2$ and join $t_1 \sqcup_{\mathbb{T}} t_2$ are:

$$\begin{aligned} &\ll a_1\gg \sqsubseteq_{\mathbb{T}} \ll a_2\gg = a_1 \sqsubseteq_{\mathbb{A}} a_2, \quad [\![c : tl_1, tr_1]\!] \sqsubseteq_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = (tl_1 \sqsubseteq_{\mathbb{T}} tl_2) \wedge (tr_1 \sqsubseteq_{\mathbb{T}} tr_2) \\ &\ll a_1\gg \sqcup_{\mathbb{T}} \ll a_2\gg = \ll a_1 \sqcup_{\mathbb{A}} a_2\gg, \quad [\![c : tl_1, tr_1]\!] \sqcup_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = [\![c : tl_1 \sqcup_{\mathbb{T}} tl_2, tr_1 \sqcup_{\mathbb{T}} tr_2]\!] \end{aligned}$$

Similarly, we compute meet $t_1 \sqcap_{\mathbb{T}} t_2$, widening $t_1 \nabla_{\mathbb{T}} t_2$, and narrowing $t_1 \triangle_{\mathbb{T}} t_2$ of two unified trees t_1 and t_2. The top is a tree with a single $\top_{\mathbb{A}}$ leaf: $\top_{\mathbb{T}} = \ll\top_{\mathbb{A}}\gg$, while the bottom is a tree with a single $\bot_{\mathbb{A}}$ leaf: $\bot_{\mathbb{T}} = \ll\bot_{\mathbb{A}}\gg$.

Transfer functions. We define lifted transfer functions for tests, forward assignments ($\text{ASSIGN}_{\mathbb{T}}$), and `#if`-s ($\text{IFDEF}_{\mathbb{T}}$) [13]. There are two types of tests: *expression-based tests*, denoted by $\text{FILTER}_{\mathbb{T}}$, that occur in `while` and `if`-s, and *feature-based tests*, denoted by $\text{FEAT-FILTER}_{\mathbb{T}}$, that occur in `#if`-s. Transfer functions $\text{ASSIGN}_{\mathbb{T}}$ and $\text{FILTER}_{\mathbb{T}}$ modify only leaf nodes, while $\text{FEAT-FILTER}_{\mathbb{T}}$ and $\text{IFDEF}_{\mathbb{T}}$ add, modify, or delete decision nodes of a decision tree. This is due to the fact that the analysis information about program variables is located in leaf nodes, while the information about features is located in decision nodes.

Transfer function $\text{ASSIGN}_{\mathbb{T}}$ for handling an assignment $\texttt{x:=}e$ in the input tree t is described by Algorithm 1. Note that `x` is a program variable, and $e \in \mathit{Exp}$ may contain only program variables. $\text{ASSIGN}_{\mathbb{T}}$ descends along the paths of the

Algorithm 2: $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta)$

```
1 switch $\theta$ do
2     case $(e_{\mathbb{F}} \bowtie e_{\mathbb{F}}) \mid\mid (\neg(e_{\mathbb{F}} \bowtie e_{\mathbb{F}}))$ do
3         $J = \texttt{FILTER}_{\mathbb{D}}(\top_{\mathbb{D}}, \theta)$; return $\texttt{RESTRICT}(t, \mathbb{K}, J)$
4     case $\theta_1 \wedge \theta_2$ do
5         return $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_1) \sqcap_{\mathbb{T}} \texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_2)$
6     case $\theta_1 \vee \theta_2$ do
7         return $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_1) \sqcup_{\mathbb{T}} \texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_2)$
```

decision tree t up to a leaf node a, where $\text{ASSIGN}_{\mathbb{A}}$ is invoked to substitute expression e for variable `x` in a. Similarly, transfer function $\text{FILTER}_{\mathbb{T}}$ for handling expression-based tests $e \in \mathit{Exp}$ is implemented by applying $\text{FILTER}_{\mathbb{A}}$ leaf-wise, so that the test e is satisfied by all leaves.

Transfer function $\text{FEAT-FILTER}_{\mathbb{T}}$ for feature-based tests θ is described by Algorithm 2. It reasons by induction on the structure of θ (we assume negation is applied to atomic propositions). When θ is an atomic constraint over numerical features (Lines 2,3), we use $\text{FILTER}_{\mathbb{D}}$ to approximate θ, thus producing a set of constraints $J \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ defined over $\mathbb{F}$, which are then added to the tree t, possibly discarding all paths of t that do not satisfy θ. This is done by calling function $\texttt{RESTRICT}(t, \mathbb{K}, J)$ that adds linear constraints from J to t in ascending order with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ (see [13, Sect. 5]). Note that θ may not be representable exactly in $\mathbb{C}_{\mathbb{D}}$ (e.g., in the case of non-linear constraints over $\mathbb{F}$), so $\text{FILTER}_{\mathbb{D}}$ may produce a set of constraints approximating it. When θ is a conjunction (resp., disjunction) of two feature expressions (Lines 4,5) (resp., (Lines 6,7)), the resulting decision trees are merged by operation meet $\sqcap_{\mathbb{T}}$ (resp., join $\sqcup_{\mathbb{T}}$).

Finally, transfer function $\text{IFDEF}_{\mathbb{T}}$ is defined as:

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

where $[\![\overline{s}]\!]_{\mathbb{T}}(t)$ is transfer function for $\overline{s}$. Transfer function $\text{ASSERT}_{\mathbb{T}}(t, \texttt{assert}(e))$ analyzes all constraints in leaf nodes $\ll a \gg$ of a tree t and replaces a with: (1) $\top$, if test e is always valid in a (i.e., $a \sqsubseteq_{\mathbb{A}} \text{FILTER}_{\mathbb{A}}(a, e)$); (2) $\bot$ if test $\neg e$ is always valid in a (i.e., $a \sqsubseteq_{\mathbb{A}} \text{FILTER}_{\mathbb{A}}(a, \neg e)$); and (3) $\times$, otherwise.

Lifted analysis. The abstract operations and transfer functions of $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ can be used to define the lifted analysis for program families. Tree t_{in} at the initial location has only one leaf node $\top_{\mathbb{A}}$ and decision nodes that define the set $\mathbb{K}$. Analysis properties are propagated forward from the first program location towards the final location taking assignments, `#if`-s, and tests into account with widening and narrowing around `while`-s. We apply delayed widening [8]. The *soundness* of the lifted analysis based on $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ follows immediately from the soundness of all operators and transfer functions of $\mathbb{D}$ and $\mathbb{A}$ (shown in [13]).

3.2 Extended Lifted Analysis

If holes in a program sketch occur in expressions that can be exactly represented in the underlying numerical domain $\mathbb{D}$, then we can handle those holes in a more efficient symbolic way by an extended lifted analysis. Given Polyhedra domain P, we say that a hole `??` can be *exactly represented* in P, if it occurs in an expression of the form: $\alpha_1 x_1 + \dots \alpha_i \texttt{??} + \dots \alpha_k x_k + \beta$, where $\alpha_1, \dots, \alpha_k, \beta \in \mathbb{Z}$ and $x_1, \dots x_k$ are program variables or other hole occurrences. Similarly, we define when a hole can be exactly represented in Interval and Octagon domains.

When a hole $\texttt{??}_i^{[n,n']}$ in a program sketch $\hat{s}$ occurs in an expression that can be represented exactly in domain $\mathbb{D}$, we eliminate `??` by using the rewrite rule:

$$s[\texttt{??}^{[n,n']}] \rightsquigarrow s[\texttt{A}] \tag{R-2}$$

where $s[\texttt{??}^{[n,n']}]$ is a basic statement and `A` is a fresh feature with domain $[n, n']$.

Example 3. The hole `??` in the HelloWorld sketch in Fig. 1 cannot be exactly represented in Polyhedra domain P, since it occurs in expression `x*??`. However, both holes in the Loop sketch in Fig. 2 can be exactly represented in P, since they occur in expressions `1*??` and `1*x-1*??` > 0. The HelloWorld family is obtained using (R-1) rule as shown in Fig. 5a, while the Loop family is given in Fig. 5d where holes are replaced with features `A` and `B` using (R-2) rule. □

After applying (R-2) rule, features can occur in arbitrary expressions in `Rewrite`$(\hat{s})$, not only in presence conditions of `#if`-s as before. Therefore, variable assignments and tests in `Rewrite`$(\hat{s})$, which may contain reads of features now, might also impact some linear constraints within decision nodes as well as some invariants within leaf nodes. Thus, we define new, extended versions of $\text{ASSIGN}_{\mathbb{T}}$ and $\text{FILTER}_{\mathbb{T}}$ that take into account possibility of features occurring in expressions. Note that $\text{ASSIGN}_{\mathbb{T}}$ and $\text{FILTER}_{\mathbb{T}}$ can now modify both leaf and decision nodes, and the analysis information about features can be located in both leaf and decision nodes. The definition of decision tree lifted domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ is slightly refined, such that the leaf abstract domain $\mathbb{A}$ is now defined over both program and feature variables $Var \cup \mathbb{F}$, while the decision node abstract domain $\mathbb{C}_{\mathbb{D}}$ remains to be defined over $\mathbb{F}$.

Assignments. Transfer function $\text{ASSIGN}_{\mathbb{T}}$ calls $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}e, \mathbb{K})$ given in Algorithm 3. It accumulates into the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (initialized to $\mathbb{K}$), constraints encountered along the paths of the decision tree (Lines 5,6), up to the leaf nodes where assignment is performed by $\text{ASSIGN}_{\mathbb{A}}$ (Line 2) and the obtained result is then restricted to satisfy the accumulated constraints C by using $\text{FILTER}_{\mathbb{A}}$ (Line 3). This is possible due to the fact that $\mathbb{A}$ is now defined over $Var \cup \mathbb{F}$.

Tests. Transfer function $\text{FILTER}_{\mathbb{T}}$ for handling tests e calls $\text{FILTER}_{\mathbb{T}}(t, e, \mathbb{K})$ described by Algorithm 4. When t is a leaf node, test e is handled using $\text{FILTER}_{\mathbb{A}}$ applied on an abstract element from $\mathbb{A}$, which is obtained by merging constraints from the leaf node and decision nodes along the path to that leaf (Line 2). The obtained result a is projected on feature variables using $\upharpoonright_{\mathbb{F}}$ to generate a new set

Algorithm 3: $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}e, C)$

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

Algorithm 4: $\texttt{FILTER}_{\mathbb{T}}(t, e, C)$

```
1 if isLeaf($t$) then
2     $a = \texttt{FILTER}_{\mathbb{A}}(t \uplus \alpha_{\mathbb{C}_{\mathbb{D}}}(C), e)$;
3     $J = \gamma_{\mathbb{C}_{\mathbb{D}}}(a \restriction_{\mathbb{F}})$;
4     if isRedundant($J, C$) then return $\ll a \gg$;
5     else return $\texttt{RESTRICT}(\ll a \gg, C, J \backslash C)$;
6 if isNode($t$) then
7     $l = \texttt{FILTER}_{\mathbb{T}}(t.l, be, C \cup \{t.c\})$;
8     $r = \texttt{FILTER}_{\mathbb{T}}(t.r, be, C \cup \{\neg t.c\})$;
9     return $[\![t.c : l, r]\!]$
```

of constraints over features $J \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (Line 3). If the constraints from J are not redundant with respect to C (this is done by checking $\alpha_{\mathbb{C}_{\mathbb{D}}}(C) \sqsubseteq_{\mathbb{D}} \alpha_{\mathbb{C}_{\mathbb{D}}}(J)$), they are added to the given path by calling $\texttt{RESTRICT}(\ll a \gg, C, J \backslash C)$ (Line 5).

Example 4. Consider program families HelloWorld and Loop in Fig. 5a and Fig. 5d. The HelloWorld family is analyzed using algorithms from the basic lifted analysis, while the Loop family using the extended lifted analysis. Figures 3 and 4 show the inferred invariants at the locations before assertions. □

3.3 Synthesis Algorithm

We can now frame the sketch synthesis problem as an lifted analysis problem. In particular, we delegate the effort of conducting an effective search of all possible hole realisations to an efficient lifted static analyzer. Once the lifted analysis of the corresponding program family is performed, we can see from the inferred decision trees in the final location for which variants the assertion is valid. Those variants that satisfy the encountered linear constraints along the valid top-down paths, represent the correct hole realisations that satisfy the final assertion.

The synthesis algorithm SYNTHESIZE($\hat{s} : Stm$) for solving a sketch $\hat{s}$ consists of the following steps: (1) Program sketch $\hat{s}$ is first encoded as a program family $\overline{s} = \texttt{Rewrite}(\hat{s})$. (2) We call function LIFT_ANALYZE($t_{in}, \overline{s}$), which takes as input the decision tree t_{in} and the program family $\overline{s}$ and returns a decision tree t in the final location of $\overline{s}$ obtained after performing the lifted analysis of $\overline{s}$. (3)

```
void main(int x){
  int z:=??1*x+??2;
  assert(z≥2*x &&
           z≤2*x+2);
}
```

Fig. 6: LinExp.

```
void main(unsigned int x){
  int y:=x;
  if (x+5>??) y := y+1;
  else y := y-1;
  assert (y≤x);
}
```

Fig. 7: Conditional.

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

Fig. 8: 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. 9: NestedLoop.

The inferred decision tree t is analyzed, and the variants $\mathbb{K}' \subseteq \mathbb{K}$ for which $\top$ ('correct') leaf nodes are found, are returned as solutions.

Theorem 2. *SYNTHESIZE($\hat{s}$) is correct and terminates.*

4 Evaluation

Implementation We have developed a prototype program synthesizer, called FamilySketcher, which is based on the tools SPLNum^{2}Analyzer [13] for analyzing `#if`-enriched C programs with numerical features and `Function` [25] for proving program termination. It uses the lifted decision tree domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, where both $\mathbb{D}$ and $\mathbb{A}$ represent numerical abstract domains (polyhedra, in our case). The abstract operations and transfer functions of the numerical polyhedra domain are provided by the APRON library [19]. The tool is written in OCaml and consists of around 7K LOC. The current front-end of the tool provides a limited support for arrays, pointers, recursion, `struct` and `union` types.

Experiment setup and Benchmarks All experiments are executed on a 64-bit Intel®Core^{TM} i7-8700 CPU@3.20GHz $\times$ 12, Ubuntu 18.04.5 LTS, with 8 GB memory, and we use a timeout value of 200 sec. All times are reported as average over five independent executions. 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_sketcher (and https://zenodo.org/record/4118540#.X7aFUWVKjIU). We compare our approach with program sketching tool `Sketch` version 1.7.6 that uses SAT-based inductive synthesis [24,23] as well as with the `Brute-Force` enumeration approach that analyzes all variants, one by one, using a single-program analysis. The evaluation is performed on several C numerical sketches collected from the `Sketch` project [24,23] and from the Syntax-Guided Synthesis Competition (https://sygus.org/) [1]. We use the following benchmarks: HelloWorld (Fig. 1), Loop (Fig. 2), LinExp (Fig. 6), Conditional (Fig. 7), LoopCond (Fig. 8), and NestedLoop (Fig. 9).

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

The Loop sketch is analyzed using the extended lifted analysis, so both holes are handled symbolically by (R-2) rule. Thus, our approach does not depend on sizes of hole domains. FamilySketcher terminates in (around) 0.007 sec for 5, 8, and 16-bits sizes of holes. In contrast, `Sketch` does depend on the sizes of holes. It terminates in 33.74 sec for 16-bits sizes, and times out for bigger sizes. Consider a variant of Loop (see Fig. 2), denoted Loop', where the assertion in location ⑦ is changed to `assert` $(\mathtt{y} < 8)$. The performance of our tool is the same as for Loop. In contrast, `Sketch` cannot resolve Loop' and fails to report a solution, since it uses only 8 unrollments of the loop by default. If the loop is unrolled 9 times, `Sketch` terminates in 0.20 sec for 5, and 2.29 sec for 16-bits sizes. FamilySketcher reports all solutions $\mathtt{A}\text{-}\mathtt{B} \geq 3$ for Loop (resp., $1 \leq \mathtt{A}\text{-}\mathtt{B} \leq 7$ for Loop'), while `Sketch` reports only one solution.

The LinExp sketch contains two holes. The first one $??_1$ is handled explicitly by (R-1) rule while the second one $??_2$ symbolically by (R-2) rule. The performance of FamilySketcher depends on the size of $??_1$. The decision tree inferred in the location before the assertion contains one leaf node for each possible value of feature `A` ($\mathtt{dom}(A) = [0, 3]$ in this case), where features `A` and `B` represent $??_1$ and $??_2$. We obtain all solutions: $\mathtt{A} = 2 \wedge 0 \leq \mathtt{B} \leq 2$. `Sketch` scales better in this case reporting one solution. Similar results we obtain for HelloWorld sketch.

The Conditional sketch contains one hole that can be handled symbolically by (R-2) rule. FamilySketcher has similar running times for all domain sizes of the hole, and reports all solutions $0 \leq \mathtt{??} \leq 4$. `Sketch`'s performance declines with the size of domains, and times out for sizes greater than 19-bits.

The LoopCond sketch contains one hole that can be handled symbolically by (R-2) rule. FamilySketcher has similar running times for all domain sizes, and reports two solutions $\mathtt{??} \in \{0, 1\}$. 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). So, `Sketch`'s performance declines with the growth of size of the hole, and times out for 16-bits. Consider a variant of LoopCond (see Fig. 8), denoted LoopCond', where one additional hole exists in `while`-guard $(\mathtt{x} \geq ??_1)$ and the assertion is changed to `assert` $(\mathtt{y} \geq 1)$. FamilySketcher reports all solutions: $??_1 \geq 0 \wedge ??_2 \geq 2$. `Sketch` performs similarly for both variants.

Finally, NestedLoop sketch contains two holes that can be handled symbolically by (R-2) rule. FamilySketcher terminates in (around) 0.05 sec for all sizes of holes. In contrast, `Brute-Force` takes 4.18 sec for 5-bit size of holes and times out for larger sizes, while `Sketch` cannot resolve this benchmark.

Discussion In summary, we can conclude 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 (R-1) rule the performances of our tool decline. However, even in this case our tool scales better than the `Brute-Force` approach.

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

Bench.	5 bits			8 bits			16 bits		
	Family Sketcher	`Sketch`	`Brute Force`	Family Sketcher	`Sketch`	`Brute Force`	Family Sketcher	`Sketch`	`Brute Force`
Loop	0.007	0.215	0.628	0.007	0.218	67.79	0.007	33.74	timeout
Loop'	0.007	0.205	0.627	0.007	0.206	60.59	0.007	2.292	timeout
LinExp	0.165	0.222	0.479	26.99	0.238	36.80	timeout	timeout	timeout
Conditional	0.002	0.210	0.019	0.002	0.210	0.155	0.004	3.856	54.68
LoopCond	0.011	0.225	0.065	0.013	0.262	0.404	0.013	timeout	191.43
LoopCond'	0.022	0.221	1.615	0.022	0.267	199.95	0.023	timeout	timeout
NestedLoop	0.053	timeout	4.186	0.054	timeout	timeout	0.054	timeout	timeout

The performances of FamilySketcher can be improved in several ways. First, many abstract operations and transfer functions can be further optimized. Second, instead of APRON we can use other efficient libraries that support numerical domains, such as ELINA [22]. Finally, by using libraries that support more expressive domains, such as non-linear constraints (e.g., polynomials, exponentials [4]), our tool will benefit and more sketches will be handled by (R-2) rule.

5 Related Work and Conclusion

The existing sketching approach `Sketch` [23,24], which uses SAT-based inductive synthesis, is more general than our approach although it is most successful for synthesizing bit-manipulating programs. `Sketch` reasons about loops by unrolling them, so is very sensitive to the degree of unrolling. Our approach does not have this constraint, as we use widening instead of fully unrolling loops, so that we can handle directly 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. `Sketch` iteratively generates a finite set of inputs and performs SAT queries to identify values for the holes. Hence, `Sketch` may need several iterations to converge reporting only one solution. In contrast, our approach needs only one iteration reporting several, and often all, solutions.

Decision-tree abstract domains have been used in abstract interpretation community recently [9,6,27]. Segmented decision tree abstract domains have been used [9,6] to enable path dependent static analysis, while Urban and Mine [27] use decision tree-based abstract domains to prove program termination.

Another way to speed up lifted analysis is via so-called variability abstractions [14,15,16], which aim tame the combinatorial explosion of the number of variants and reduce it to something more tractable. It would be interesting to apply the obtained abstract lifted analysis for resolving program sketches.

To conclude, in this work we employ techniques from abstract interpretation and product-line analysis for automatic resolving of program sketches.

References

1. Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013*, pages 1–8. IEEE, 2013.
2. Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer, 2013.
3. Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th Inter. Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.
4. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *LNCS*, pages 1349–1361. Springer, 2005.
5. Milan Ceska, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter 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*, volume 11500 of *LNCS*, pages 107–125. Springer, 2019.
6. Junjie Chen and Patrick Cousot. A binary decision tree abstract domain functor. In *Static Analysis - 22nd International Symposium, SAS 2015, Proceedings*, volume 9291 of *LNCS*, pages 36–53. Springer, 2015.
7. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.
8. Patrick Cousot and Radhia 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*, pages 238–252. ACM, 1977.
9. Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A scalable segmented decision tree abstract domain. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010.
10. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on POPL'78*, pages 84–96. ACM Press, 1978.
11. Aleksandar S. Dimovski. Lifted static analysis using a binary decision diagram abstract domain. In *Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019*, pages 102–114. ACM, 2019.
12. Aleksandar S. Dimovski. On calculating assertion probabilities for program families. *Prilozi Contributions, Sec. Nat. Math. Biotech. Sci, MASA*, 41(1):13–23, 2020.
13. Aleksandar S. Dimovski, Sven Apel, and Axel 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*, volume 12649 of *LNCS*, pages 67–86. Springer, 2021.
14. Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
15. Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Finding suitable variability abstractions for family-based analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 217–234. Springer, 2016.

16. Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Finding suitable variability abstractions for lifted analysis. *Formal Asp. Comp.*, 31(2):231–259, 2019.
17. Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482, 2016.
18. Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Effective analysis of C programs by rewriting variability. *Art Sci. Eng. Program.*, 1(1):1, 2017.
19. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.
20. Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015.
21. David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.
22. Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In *Proceedings of the 36th ACM SIGPLAN Conference on PLDI, 2015*, pages 303–313. ACM, 2015.
23. Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
24. Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 281–294. ACM, 2005.
25. Caterina 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*, volume 9035 of *LNCS*, pages 464–466. Springer, 2015.
26. Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs.* PhD thesis, École Normale Supérieure, Paris, France, 2015.
27. Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In *Static Analysis - 21st International Symposium, SAS 2014. Proceedings*, volume 8723 of *LNCS*, pages 302–318. Springer, 2014.
28. Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 27(4):18:1–18:33, 2018.