Lifted Termination Analysis by Abstract Interpretation and Its Applications

Aleksandar S. Dimovski
aleksandar.dimovski@unt.edu.mk
Mother Teresa University
Skopje, North Macedonia

Abstract

This paper is focused on proving termination for program families with numerical features by using abstract interpretation. Furthermore, we present an interesting application of the above lifted termination analysis for resolving "sketches", i.e. partial programs with missing numerical parameters (holes), such that the resulting complete programs always terminate. To successfully address the above problems, we employ an abstract interpretation-based framework for inferring sufficient preconditions for termination of single programs that synthesizes piecewise-defined ranking functions.

We introduce a novel lifted decision tree domain for termination, in which decision nodes contain linear constraints defined over numerical features and leaf nodes contain piecewise ranking functions defined over program variables. Moreover, we encode a program sketch as a program family, thereby allowing the use of the lifted termination analysis as a program sketcher. In particular, we aim to find the variants (family members) that terminate under all possible inputs, which represent the correct sketch realizations.

We have implemented an experimental lifted termination analyzer, called SPLFuncTion, for proving termination of `#if`-based C program families and for termination-directed resolving of C program sketches. We have evaluated our approach by a set of loop benchmarks from SV-COMP, and experimental results confirm the effectiveness our approach.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; *Software creation and management*; • **Theory of computation** → Semantics and reasoning.

Keywords: Program families, Lifted termination analysis, Program sketching, Abstract interpretation

GPCE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487202

ACM Reference Format:
Aleksandar S. Dimovski. 2021. Lifted Termination Analysis by Abstract Interpretation and Its Applications. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486609.3487202

1 Introduction

A *program family* (software product line) describes a set of similar programs as *variants* of some common code base [1]. This enables end users to derive a program suitable to a particular application scenario. The well-known `#if` directives from the C preprocessor CPP represent the most common mechanism to implement such program families [22]. Optional program fragments are annotated with `#if` directives, which are included or excluded at compile-time, depending on the functionality that end users specify. Program families are quite popular in certain application domains, such as cars, phones, medicine, robotics, etc [1].

Static analyses are a powerful tool to prove the correctness of programs, and so it is very important to apply static analyses to program families [32]. However, the traditional static analyses cannot be directly applied to program families as they only analyze pre-processed single programs. To handle large program families, researchers have developed the so-called *lifted* (*variability-aware*) *static analyses*, which process the common code base directly, exploiting similarities among individual variants to reduce analysis effort [3, 4, 11, 27, 37]. They analyze common and variable parts of the common code base only once and are able to infer analysis properties in all valid variants of the program family. Automatic proving of program termination is a fundamental problem in program analysis. Termination bugs can compromise programs by making them irresponsive. They can also be exploited in denial-of-service attacks [25]. Therefore, proving program termination is important for establishing program reliability. To address this problem in practice, several static analysis techniques have been proposed [5, 19, 35]. They use different strategies to synthesize *ranking functions*, a well-founded metric which strictly decreases during program execution.

In this work, we introduce an approach to *lift* an existing single-program termination analysis based on abstract interpretation [34–36] to program families. *Abstract interpretation*

[8, 29] is a general theory for approximating the semantics of programs. It represents a powerful technique for deriving approximate, albeit computable static analyses, by using fully automatic algorithms. In particular, we extend the traditional termination analysis developed by Urban and Mine for automatically inferring sufficient preconditions that a given program will always terminate under all inputs. This is done by employing abstract interpretation techniques to determine piecewise-defined ranking functions, which provide upper bounds on the number of execution steps to termination as a function of the program variables. We lift this analysis to a lifted termination analysis that operates on the entire program family in one single pass. The elements of the lifted domain are *two-level lifted decision trees*, in which the first-level decision nodes are labelled with linear constraints over feature variables, the second-level decision nodes are labelled with linear constraints over program variables, and the leaf nodes are affine functions over program variables. In fact, the second-level decision nodes and leaf nodes represent piecewise-defined ranking functions. The lifted decision trees recursively partition the space of all possible valid variants, while the piecewise-defined ranking functions provide termination-related information corresponding to each partition. The efficiency of our decision tree-based lifted analysis comes from the opportunity to share equal subtrees, in case some termination properties are independent from the values of some features. This possibility for sharing is especially important in the case of program families that contain numerical features with large domains, thus giving rise to astronomical configuration spaces.

Several other approaches can also be used for termination analysis of program families. First, the simplest brute-force approach uses a preprocessor to generate all variants of a family and then applies an existing off-the-shelf single-program analyzer to each individual variant, one-by-one. This approach is shown to be very inefficient [4, 37]. Second, the tuple-based lifted analysis maintains one analysis property (ranking function) per variant in tuples. However, this explicit enumeration in tuples becomes computationally intractable with larger program families because the number of variants grows exponentially (or even faster) with the number of features. Third, the variability encoding approach [20, 38] replaces compile-time variability with run-time variability (non-determinism) . In particular, a given program family is transformed into a single program by encoding features with ordinary program variables that are non-deterministically initialized to any value from their domain and by encoding `#if` directives with conditional `if` statements. The resulting single programs, called *variability simulators*, can be analyzed using off-the-shelf single-program analyzers. However, the shortcoming of this approach is that it does not consider disjunctive properties arising from features as lifted analyses do consider. This leads to imprecisions and rough approximations in the inferred analysis invariants, which often results in failures of showing the required program properties.

Additionally, we leverage the lifted termination analysis to synthesize numerical program sketches that always terminate. A *sketch* is a partial program with missing numerical expressions called *holes* to be discovered by the synthesizer. Previous approaches for program sketching [30, 31] automatically synthesize integer constant values for the holes so that the resulting complete program satisfies *safety properties* ("something bad never happens") in the form of assertions. This is called an *assertion-directed program sketching*. In this work, we consider program sketching with respect to liveness properties ("something good eventually happens") in the form of termination. In particular, we construct a program synthesizer that automatically finds integer values for the holes so that the resulting complete program *terminates* under all possible inputs. This is done by using our lifted termination analysis of program families. We refer to this as *termination-directed program sketching*. The key observation is that all possible sketch realizations constitute a program family, where each numerical hole is represented as a numerical feature. Hence, we reduce the termination-directed program sketching problem to selecting those variants from the corresponding program family that always terminate. This can be efficiently done by using our lifted termination analysis, which is able to converge to a solution very fast even for program families with astronomical sizes. This is particularly true for sketches in which holes appear in linear expressions that can be exactly represented in the numerical domains used in the lifted decision trees (e.g., polyhedra). In those cases, we can design more efficient lifted analysis with extended transfer functions for assignments and tests, which can directly handle *read* accesses to the feature variables. To the best of our knowledge, this is the first work that tackles the problem of termination-directed program sketching.

We have implemented our approach in a prototype lifted termination analyzer, called SPLFuncTion, which is built on top of the termination analyzer FuncTion [34, 35]. The numerical abstract domains, including intervals, octagons, polyhedra, from the APRON library [21] are used as parameters to implement decision and leaf nodes of the underlying decision trees. We illustrate this approach for automatically inferring sufficient preconditions for termination of `#if`-annotated C program families and for automatically completing various numerical C sketches. We evaluate our approach on a selected set of benchmarks from SV-COMP suite [1]. We compare its time performance against the tuple-based lifted analysis and its precision performance against the variability encoding approach.

In summary, we make several contributions in this work: (1) We propose a novel lifted termination analysis based on

[1] Int. Comp. on Software Verification (https://sv-comp.sosy-lab.org/2020/).

```
void main(){
①  int x, y;
②  #if (A<B) y = 50; #else y = 51; #endif
③  while ④ (x ≥ 0) {
⑤    if (y ≤ 50) x = x-1; else x = x+1;
⑥    #if (A<2) y=y-1; #else y=y+1 #endif }
}
```

Figure 1. The program family `cav2006.c`.

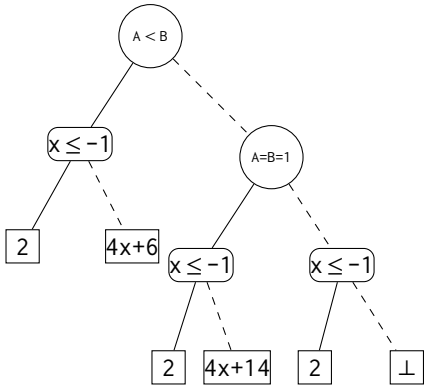

Figure 2. Inferred lifted decision tree at location ① of `cav2006.c` (solid edges = true, dashed edges = false, circles = first-level decision nodes, rounded rectangles = second-level decision nodes, rectangles = leaves).

abstract interpretation for program families with numerical features; (2) We show the applications of the lifted termination analysis for resolving termination-directed program sketches; (3) We implement a prototype lifted analyzer, called SPLFuncTion, which performs termination analysis of `#if`-annotated C programs and resolves C program sketches; and (4) We evaluate our approach on benchmarks from SV-COMP by comparing its performance against the tuple-based lifted analysis and variability encoding.

2 Motivating Examples

Let us consider the code base of `cav2006.c` program family given in Fig. 1. It represents a variability-extended version of the program `GopanReps-CAV2006.c` from the category `termination-crafted-lit` of the SV-COMP suite. The set of features is $\mathcal{F} = \{\mathtt{A}, \mathtt{B}\}$, where `A` and `B` are numerical features whose domains are $[0, 3]$ and $[1, 2]$, respectively. The set of valid configurations includes all possible combinations of feature's values, thus $\mathcal{K} = \{(\mathtt{A}=0) \wedge (\mathtt{B}=1), (\mathtt{A}=1) \wedge (\mathtt{B}=1), (\mathtt{A}=2) \wedge (\mathtt{B}=1), (\mathtt{A}=3) \wedge (\mathtt{B}=1), (\mathtt{A}=0) \wedge (\mathtt{B}=2), (\mathtt{A}=1) \wedge (\mathtt{B}=2), (\mathtt{A}=2) \wedge (\mathtt{B}=2), (\mathtt{A}=3) \wedge (\mathtt{B}=2)\}$. For each configuration from $\mathcal{K}$, a different variant is derived by appropriately resolving the two `#if` directives in the code base, depending on how features are set at compile-time. For example, the variant corresponding to configuration $(\mathtt{A}=0) \wedge (\mathtt{B}=1)$ will have `A` and `B` set to 0 and 1, so that the assignments `y = 50` and `y = y-1` will be included in this variant.

Assume that we want to perform *lifted termination analysis* of `cav2006.c` using the *polyhedra* numerical domain [9] as parameter. If we use the *lifted decision tree domain* proposed in this work, then the inferred decision tree in the initial program location ①, representing the ranking function of `cav2006.c` (since this is a backward analysis), is depicted in Fig. 2. Notice that the first-level decision nodes of the decision tree in Fig. 2 are labeled with *polyhedra* linear constraints over features (`A` and `B`), the second-level decision nodes are labeled with *polyhedra* linear constraints over program variables (`x` and `y`), while the leaves are labeled with affine functions over program variables. In figures, we use circles to represent first-level decision nodes, rounded rectangles to represent second-level decision nodes, and rectangles to represent leaves. The edges 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 the first-level decision nodes partition the space of valid configurations $\mathcal{K}$ (i.e. the possible values of features) and the second-level decision nodes partition the memory space (i.e. the possible values of program variables), we implicitly take into account domains of features and program variables. For example, the node with constraint $(\mathtt{A}<\mathtt{B})$ is satisfied when $(\mathtt{A}<\mathtt{B}) \wedge (0 \leq \mathtt{A} \leq 3) \wedge (1 \leq \mathtt{B} \leq 2)$. We can see that decision trees offer possibilities for sharing of analysis equivalent information corresponding to different configurations, thus they provide symbolic and compact representation of lifted analysis elements. For example, the decision tree in Fig. 2 shows that: (1) when $(\mathtt{A}<\mathtt{B})$ is true the variants terminate with ranking function: 2 if $(\mathtt{x} \leq -1)$ and 4x+6 if $(\mathtt{x} \geq 0)$; (2) when $(\mathtt{A}=\mathtt{B}=1)$ holds the ranking function is: 2 if $(\mathtt{x} \leq -1)$ and 4x+14 if $(\mathtt{x} \geq 0)$; (3) when $(\mathtt{A} \geq \mathtt{B}) \wedge (\mathtt{A} \neq 1)$ holds the variants potentially not-terminate ($\perp$ answer) for $(\mathtt{x} \geq 0)$. In the case (3), the `else` branches of both `$if`-s are taken making $(\mathtt{y} \geq 51)$ to hold in the `while`-body. Therefore, the `else` branch `x = x+1` in location ⑤ will always be taken, thus causing the non-termination for $(\mathtt{x} \geq 0)$. On the other hand, the case (2) will need two iterations more to terminate than the case (1) for $(\mathtt{x} \geq 0)$, due to the fact when $(\mathtt{A}=\mathtt{B}=1)$ the initial value of `y` is 51 thus causing `x = x+1` in location ⑤ to be executed in the first iteration of `while` and after this iteration we will have $(\mathtt{y} \leq 50)$ making `x = x-1` to be executed in all following iterations. This is reflected in the ranking functions of cases (1) and (2), where the variants $(\mathtt{A}=\mathtt{B}=1)$ need 8 execution steps (2 `while`-iterations) more to terminate than $(\mathtt{A}<\mathtt{B})$. In effect, the decision tree-based representation uses only three ranking functions, although there are eight variants in total. This ability for sharing is the key motivation behind the efficiency of this representation.

```
void main() {
①  int x;
②  int y = ??1;
③  while (x ≤ 10) {
④      if (y ≥??2) x = x+1; else x = x-1; }
}
```

Figure 3. The sketch `loop1.c`

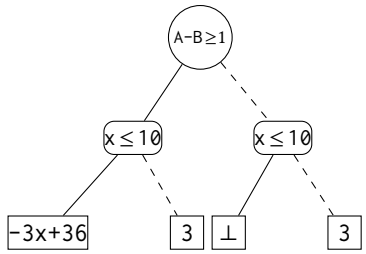

Figure 4. Inferred lifted decision tree at location ① of `loop1.c` (features A and B correspond to holes $??_1$ and $??_2$).

Alternatively, by using variability encoding the program family `cav2006.c` can be transformed into a single program, called variability simulator [38], which can be analyzed using the single-program termination domain [34, 35]. In particular, the variability simulator of `cav2006.c` is obtained by encoding features as ordinary program variables that are non-deterministically initialized, that is: `int A = [0,3], B = [1,2]`, and by encoding `#if` directives as ordinary `if` statements. The ranking function inferred in this case is: 5 if $(x \leq -1)$ and $\bot$ if $(x \geq 0)$. However, this invariant is not strong enough to establish the termination of variants when $(A<B)$or$(A=B=1)$.

As observed before in the literature [6, 16], a sketch can be represented as a program family, such that all possible realizations of the sketch correspond to possible variants in the program family. For example, `loop1.c` sketch, given in Fig. 3, contains two numerical holes. It can be encoded as a program family, in which the two holes $??_1$ and $??_2$ are replaced by two numerical features A and B with domain $[\mathtt{Min}, \mathtt{Max}] \subseteq \mathbb{Z}$ [2]. Hence, there are $(\mathtt{Max} - \mathtt{Min} + 1)^2$ variants that can be instantiated from the `loop1.c` program family. Notice that features in such a program family can occur in arbitrary expressions, not only in presence conditions of `#if`-s as before in traditional program families. Therefore, we define an extended lifted termination analysis that can handle such program families (features occur in arbitrary expressions). If we analyze `loop1.c` program family using the extended lifted termination analysis, we obtain as result the lifted decision tree shown in Fig. 4. We can see that the ranking function is *fully-defined* (over all possible values of x) for variants satisfying $(A \geq B+1)$. 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 `loop1.c` sketch. Note that when $\neg(A \geq B+1)$ holds we obtain potential not-termination ($\bot$ answer) for $(x \leq 10)$.

3 A Language for Program Families

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 $A \in \mathcal{F}$, i.e. $k(A) \in \mathrm{dom}(A)$. 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 be represented by a 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 $\mathcal{F}$:

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

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

We consider a simple sequential non-deterministic programming language, which will be used to exemplify our work. The program variables $Var = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. To encode multiple variants, a new compile-time conditional statement is included: "`#if` (θ) s `#endif`" contains a feature expression θ as a presence condition, such that only if θ is satisfied by $k \in \mathcal{K}$ the statement s will be included in the variant corresponding to k. The syntax is:

$$\begin{aligned} s &::= \mathtt{skip} \mid \mathtt{x} := ae \mid s; s \mid \mathtt{if}\ (be)\ \mathtt{then}\ s\ \mathtt{else}\ s \\ &\quad \mid \mathtt{while}\ (be)\ \mathtt{do}\ s \mid \mathtt{\#if}\ (\theta)\ s\ \mathtt{\#endif}, \\ ae &::= n \mid [n, n'] \mid \mathtt{x} \mid ae \oplus ae, \\ be &::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be \end{aligned}$$

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, x over program variables Var, $\oplus \in \{+, -, *, /\}$, and $\bowtie \in \{<, \leq, =, \neq\}$. Integer intervals $[n, n']$ denote a random integer in the interval. The set of all statements s is denoted by Stm; the set of all arithmetic expressions ae is denoted by $AExp$; and the set of all boolean expressions be is denoted by $BExp$. Any other preprocessor conditional constructs can be desugared and represented only by `#if` construct. For example, `#if` (θ) s_0 `#elif` (θ') s_1 `#endif` is translated into `#if` (θ) s_0 `#endif ; #if` $(\neg\theta \wedge \theta')$ s_1 `#endif`.

[2] Note that `Min` and `Max` represent some minimal and maximal representable integers. For example, `Min` = 0 and `Max` = 31 for 5-bit sizes of holes.

A program family is evaluated in two stages. First, the C preprocessor CPP takes a program family s and a configuration $k \in \mathcal{K}$ as inputs, and produces a variant (single program without #if-s) corresponding to k as the output. Second, the obtained variant is evaluated using the standard single-program semantics [35]. The first stage is specified by the projection function $\mathtt{P}_k$, which is an identity for all basic statements and recursively pre-processes all substatements of compound statements. Hence, $\mathtt{P}_k(\mathtt{skip}) = \mathtt{skip}$ and $\mathtt{P}_k(s;s') = \mathtt{P}_k(s);\mathtt{P}_k(s')$. For "#if ($\theta$) s #endif", statement s is included in the variant if $k \models \theta$, [3] that is:

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

4 Lifted Decision Tree Domain for Termination

Lifted analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. They directly analyze program families without preprocessing them by taking the variability of program families into account. In this work, we want to lift the termination analysis based on abstract interpretation [35, 36], which is used to infer piecewise-defined ranking functions in all program locations. The elements of the (single-program) termination analysis domain are represented by *decision trees*, where the decision nodes are labelled with linear constraints over program variables and the leaf nodes belong to an abstract domain for affine functions defined over program variables. These functions provide upper bounds on the number of execution steps to termination from a given program location. Lifted analysis for program families relies on a lifted domain of decision trees [15]. Hence, the elements of the lifted termination analysis domain are so-called *lifted decision trees*, where the leaf nodes belong to the termination domain and the decision nodes are linear constraints over feature variables. This way, we construct *two-level decision trees*, where each top-down path represents a subset of configurations $\mathcal{K}$ that satisfy the constraints over feature variables encountered along the first-level of the path and a subset of program states that satisfy the constraints over program variables encountered along the second-level of the path.

4.1 Numerical Domains

We assume that a single-program numerical domain $\mathbb{D}$ defined over a set of variables $V = \{X_1, \ldots, X_l\}$ is equipped with sound operators for concretization $\gamma_{\mathbb{D}}$ (which assigns to each abstract element from $\mathbb{D}$ a concrete meaning that represents a set of states mapping each variable from V to its value from $\mathbb{Z}$), ordering $\sqsubseteq_{\mathbb{D}}$, join $\sqcup_{\mathbb{D}}$, meet $\sqcap_{\mathbb{D}}$, bottom $\bot_{\mathbb{D}}$, top $\top_{\mathbb{D}}$, widening $\nabla_{\mathbb{D}}$, and narrowing $\triangle_{\mathbb{D}}$, as well as sound transfer functions for tests (boolean expressions) $\text{FILTER}_{\mathbb{D}}$, forward assignments $\text{ASSIGN}_{\mathbb{D}}$, and backward assignments $\text{B-ASSIGN}_{\mathbb{D}}$. More specifically, transfer function $\text{FILTER}_{\mathbb{D}}(d : \mathbb{D}, be : \mathit{BExp})$ returns an abstract element from $\mathbb{D}$ obtained by restricting d to satisfy test be; $\text{ASSIGN}_{\mathbb{D}}(d : \mathbb{D}, \mathtt{x:=}ae : \mathit{Stm})$ returns an updated version of d by abstractly evaluating $\mathtt{x:=}ae$ in it; whereas $\text{B-ASSIGN}_{\mathbb{D}}(r : \mathbb{D}, \mathtt{x:=}ae : \mathit{Stm})$ returns an abstract element from $\mathbb{D}$ such that by abstractly evaluating $\mathtt{x:=}ae$ in it produces the abstract element r. Note that r represents an invariant in the final location of the assignment $\mathtt{x:=}ae$ that needs to be propagated backwards.

In practice, the domain $\mathbb{D}$ will be instantiated with some of the known numerical domains, such as Intervals $\langle I, \sqsubseteq_I \rangle$ [8], Octagons $\langle O, \sqsubseteq_O \rangle$ [28], and Polyhedra $\langle P, \sqsubseteq_P \rangle$ [9]. 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 $X_1, \ldots X_k \in V, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}$. We will sometimes write $\mathbb{D}_V$ to explicitly denote the set of variables V over which domain $\mathbb{D}$ is defined. In this work, we will use domains $\mathbb{D}_{\mathit{Var}}$ and $\mathbb{D}_{\mathcal{F}}$ that are defined over program and feature variables, respectively. We refer to [29] for a precise definition of all abstract operations and transfer functions for Intervals, Octagons, and Polyhedra.

4.2 Function Domains for Leaf Nodes

The elements of the domain for *affine functions* $\mathbb{F}_A$ are:

$$\mathbb{F}_A = \{\bot, \top\} \cup \{f : \mathbb{Z}^{|\mathit{Var}|\to\mathbb{N}} \mid f(x_1,\ldots,x_n) = m_1 x_1 + \ldots + m_n x_n + q\}$$

where an element $f \in \mathbb{F}_A$ is a natural-valued function of program variables representing an upper bound on the number of steps to termination; the element $\bot$ represents potential non-termination; and the element $\top$ represents the lack of information to conclude (i.e. the approximations prevent the analyzer from giving a definite answer). The function $f \in \mathbb{F}_A$ represents a piece of a partially-defined ranking function over program variables from *Var*. The leaf nodes belonging to $\mathbb{F}_A \backslash \{\bot, \top\}$ and $\{\bot, \top\}$ represent *defined* and *undefined* leaf nodes, respectively. All abstract operations and transfer functions of $\mathbb{F}_A$ are defined in [35, 36].

4.3 Linear Constraints Domain for Decision Nodes

We introduce a family of abstract domains for finite sets of linear constraints $\mathbb{C}_{\mathbb{D}_V}$ defined over variables V, which are parameterized by a numerical domain $\mathbb{D}_V$. For example, the set of *polyhedral constraints* is $\mathbb{C}_{P_V} = \{\alpha_1 X_1 + \ldots + \alpha_l X_l + \beta \geq 0 \mid X_1, \ldots X_l \in V, \alpha_1, \ldots, \alpha_l, \beta \in \mathbb{Z}, \gcd(|\alpha_1|, \ldots, |\alpha_l|, |\beta|) = 1\}$. The correspondence between the set of linear constraints $\mathbb{C}_{\mathbb{D}_V}$ and the numerical domain $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$ is given using the Galois connection $\langle \mathcal{P}(\mathbb{C}_{\mathbb{D}_V}), \subseteq \rangle \underset{\alpha_{\mathbb{C}_{\mathbb{D}}}}{\overset{\gamma_{\mathbb{C}_{\mathbb{D}}}}{\leftrightarrows}} \langle \mathbb{D}_V, \sqsubseteq_{\mathbb{D}} \rangle$, where $\mathcal{P}(\mathbb{C}_{\mathbb{D}_V})$ is the power set of $\mathbb{C}_{\mathbb{D}_V}$. The abstraction function maps a set of constraints in $\mathcal{P}(\mathbb{C}_{\mathbb{D}_V})$ to a conjunction of

[3] Since $k \in \mathcal{K}$ is a valuation function, either $k \models \theta$ holds or $k \not\models \theta$ holds for any $\theta \in \mathit{FeatExp}(\mathcal{F})$.

constraints from $\mathbb{D}_V$, while the concretization function $\gamma_{\mathbb{C}_\mathbb{D}}$ maps a conjunction of constraints from $\mathbb{D}_V$ to a set of constraints in $\mathcal{P}(\mathbb{C}_{\mathbb{D}_V})$. Hence, $\mathcal{P}(\mathbb{C}_{\mathbb{D}_V})$ and $\mathbb{D}_V$ are two different (set-theoretic and logic-theoretic) representations of linear constraints over variables V.

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

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

The negation of linear constraints is formed as: $\neg(\alpha_1 X_1 + \ldots \alpha_n X_n + \beta \geq 0) = -\alpha_1 X_1 - \ldots - \alpha_n X_n - \beta - 1 \geq 0$. 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 with respect to $<_{C_\mathbb{D}}$ between c and $\neg c$.

4.4 Termination Decision Tree Domain

We now recall the termination decision tree domain [35, 36] for representing partial ranking functions. A *termination decision tree* $t' \in \mathbb{T}^T(\mathbb{C}_{\mathbb{D}_{Var}}, \mathbb{F}_A)$ over the sets $\mathbb{C}_{\mathbb{D}_{Var}}$ of linear constraints defined over program variables Var and the domain of affine functions $\mathbb{F}_A$ is: either a leaf node $\ll f \gg$ with $f \in \mathbb{F}_A$, or $[\![c' : tl', tr']\!]$, where $c' \in \mathbb{C}_{\mathbb{D}_{Var}}$ (denoted by $t'.c$) is the smallest constraint with respect to $<_{\mathbb{C}_{\mathbb{D}_{Var}}}$ appearing in the tree t' (i.e., all constraints within decision nodes in tl' and tr' are larger than c'), 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 (those that satisfy the encountered constraints), and the leaf nodes represent the partially defined ranking functions over the corresponding program states.

The termination domain $\mathbb{T}^T$ is equipped with sound operators for concretization $\gamma_{\mathbb{T}^T}$ (which assigns to each decision tree t' a concrete meaning that represents a ranking function mapping each program state to the number of execution steps to termination), ordering $\sqsubseteq_{\mathbb{T}^T}$, join $\sqcup_{\mathbb{T}^T}$, meet $\sqcap_{\mathbb{T}^T}$, bottom $\bot_{\mathbb{T}^T}$, top $\top_{\mathbb{T}^T}$, widening $\nabla_{\mathbb{T}^T}$, as well as sound transfer functions for tests $\text{FILTER}_{\mathbb{T}^T}$ and backward assignments $\text{B-ASSIGN}_{\mathbb{T}^T}$. We refer to [35, 36] for their definitions.

4.5 Lifted Decision Tree Domain

We now define the lifted decision tree domain for representing lifted ranking functions. A *lifted decision tree* $t \in \mathbb{T}^L(\mathbb{C}_{\mathbb{D}_\mathcal{F}}, \mathbb{T}^T(\mathbb{C}_{\mathbb{D}_{Var}}, \mathbb{F}_A))$ over the sets $\mathbb{C}_{\mathbb{D}_\mathcal{F}}$ of linear constraints defined over feature variables $\mathcal{F}$ and the termination decision trees $\mathbb{T}^T$ is: either a leaf node $\ll t' \gg$ with $t' \in \mathbb{T}^T$, or $[\![c : tl, tr]\!]$, where $c \in \mathbb{C}_{\mathbb{D}_\mathcal{F}}$ (denoted by $t.c$) is the smallest constraint with respect to $<_{\mathbb{C}_{\mathbb{D}_\mathcal{F}}}$ 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 configurations (those that satisfy the encountered constraints), and the leaf nodes represent the termination decision trees (i.e., partial ranking functions) for the corresponding configurations.

Algorithm 1: $\texttt{B-ASSIGN}_{\mathbb{T}^L}(t, \texttt{x:=}ae)$

```
1 if isLeaf(t) then return ≪B-ASSIGN_{T^T}(t, x:=ae)≫;
2 return [[t.c : B-ASSIGN_{T^L}(t.l, x:=ae), B-ASSIGN_{T^L}(t.r, x:=ae)]];
```

Algorithm 2: $\texttt{FILTER}_{\mathbb{T}^L}(t, be)$

```
1 if isLeaf(t) then return ≪FILTER_{T^T}(t, be)≫;
2 return [[t.c : FILTER_{T^L}(t.l, be),FILTER_{T^L}(t.r, be)]];
```

The lifted operations for $\mathbb{T}^L(\mathbb{C}_{\mathbb{D}_\mathcal{F}}, \mathbb{T}^T(\mathbb{C}_{\mathbb{D}_{Var}}, \mathbb{F}_A))$ rely on the algorithm for *tree unification* [15], which finds a common labelling of decision nodes of two trees. Hence, the result of tree unification are decision trees with the same decision nodes. All operations are then performed leaf-wise on the unified decision trees by applying the corresponding operations of domain $\mathbb{T}^T$ on leaf nodes. For example, the ordering $t_1 \sqsubseteq_\mathbb{T} t_2$ of two unified decision trees t_1 and t_2 is defined as:

$$\ll t'_1 \gg \sqsubseteq_{\mathbb{T}^L} \ll t'_2 \gg = t'_1 \sqsubseteq_{\mathbb{T}^T} t'_2,$$
$$[\![c : tl_1, tr_1]\!] \sqsubseteq_{\mathbb{T}^L} [\![c : tl_2, tr_2]\!] = (tl_1 \sqsubseteq_{\mathbb{T}^L} tl_2) \wedge (tr_1 \sqsubseteq_{\mathbb{T}^L} tr_2)$$

We refer to [15] for similar lifted operations defined for the lifted decision tree domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}_\mathcal{F}}, \mathbb{D}_{Var})$ [15], which is used for the (forward) lifted numerical analysis.

We now describe lifted transfer functions for boolean expression-based tests $be \in BExp$ ($\text{FILTER}_{\mathbb{T}^L}$), feature-based tests $\theta \in FeatExp(\mathcal{F})$ ($\text{F-FILTER}_{\mathbb{T}^L}$), and backward assignments $\texttt{x:=}ae$ ($\text{B-ASSIGN}_{\mathbb{T}^L}$). Note that the analysis information about program variables is located only in leaf nodes $t' \in \mathbb{T}^T$, whereas the analysis information about feature variables is located only in (first-level) decision nodes of lifted decision trees $c \in \mathbb{C}_{\mathbb{D}_\mathcal{F}}$. Therefore, $\text{FILTER}_{\mathbb{T}^L}$ and $\text{B-ASSIGN}_{\mathbb{T}^L}$ modify only leaf nodes belonging to $\mathbb{T}^T$, whereas $\text{F-FILTER}_{\mathbb{T}^L}$ modifies only the decision nodes belonging to $\mathbb{C}_\mathbb{D}$ domain.

The lifted transfer function $\text{B-ASSIGN}_{\mathbb{T}^L}$ for handling an assignment $\texttt{x:=}ae$, where $\texttt{x}$ is a program variable and $ae \in AExp$ contains only program variables, is described by Algorithm 1. $\text{B-ASSIGN}_{\mathbb{T}^L} : \mathbb{T}^L \times Stm \to \mathbb{T}^L$ descends along the paths of the lifted decision tree t up to a leaf node $\ll t' \gg$, where $\text{B-ASSIGN}_{\mathbb{T}^T}(t', \texttt{x:=}ae)$ [35, 36] is invoked to handle backward assignments within the termination decision tree t'. The lifted transfer function $\text{FILTER}_{\mathbb{T}^L} : \mathbb{T}^L \times BExp \to \mathbb{T}^L$ for handling boolean expression-based tests $be \in BExp$, where be contains only program variables, is implemented by Algorithm 2 by applying $\text{FILTER}_{\mathbb{T}^T}$ leaf-wise, so that the test be is satisfied by all leaves.

Algorithm 3: $\texttt{F-FILTER}_{\mathbb{T}^L}(t, \theta)$

```
1 switch θ do
2     case (e_F ⋈ e_F) || ¬(e_F ⋈ e_F) do
3         C = FILTER_{D_F}(⊤_{D_F}, θ)
4         return RESTRICT(t, K, C)
5     case θ1 ∧ θ2 do
6         return F-FILTER_{T^L}(t, θ1) ⊓_{T^L} F-FILTER_{T^L}(t, θ2)
7     case θ1 ∨ θ2 do
8         return F-FILTER_{T^L}(t, θ1) ⊔_{T^L} F-FILTER_{T^L}(t, θ2)
```

Lifted transfer function F-FILTER$_{\mathbb{T}^L} : \mathbb{T}^L \times \mathit{FeatExp}(\mathcal{F}) \to \mathbb{T}^L$ for feature-based tests $\theta \in \mathit{FeatExp}(\mathcal{F})$, where θ contains only feature variables, is described by Algorithm 3 by reasoning inductively on the structure of θ (we assume negation is only applied to atomic constraints). When θ is an atomic constraint over numerical features (Lines 2,3), we use FILTER$_{\mathbb{D}}$ to approximate θ, thus producing a set of linear constraints $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}})$, which are subsequently added to the tree t, possibly discarding all paths of t that do not satisfy θ. This is done by calling the function RESTRICT$(t, \mathcal{K}, C)$ that adds linear constraints from C to t (see [15] for a precise definition). 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 $\sqcup_{\mathbb{T}}$). The transfer function for `#if`-s is then defined as:

$$[\![\texttt{\#if}\ (\theta)\ s\ \texttt{\#end}]\!]_{\mathbb{T}^L} t = [\![s]\!]_{\mathbb{T}^L}(\texttt{F-FILTER}_{\mathbb{T}^L}(t, \theta)) \sqcup_{\mathbb{T}^L} \texttt{F-FILTER}_{\mathbb{T}^L}(t, \neg\theta)$$

where $[\![s]\!]_{\mathbb{T}^L}(t)$ is the transfer function for statement s.

4.6 Lifted Termination Analysis

The operations and transfer functions of $\mathbb{T}^L(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{T}^T)$ can be used to analyze termination of program families. The lifted termination analysis derived from $\mathbb{T}^L$ is a pure backward analysis that infers ranking functions in all program locations. The lifted transfer function $[\![s]\!]_{\mathbb{T}^L}$ of a statement s takes as input a lifted decision tree t corresponding to the final location of s, and outputs a lifted decision tree over-approximating the ranking function corresponding to the initial location of s. The input lifted decision tree t_{in} at the final location of a program has only one leaf node $\lambda x_1, \ldots, x_n.0$ (that is, the zero function), and decision nodes define the set $\mathcal{K}$. The lifted decision tree describes that in the final location of all variants there are zero execution steps to termination. Analysis properties are propagated backward from the final program location towards the initial location taking assignments, `#if`-s, and tests into account with widening around `while`-s. We apply delayed widening [8], which means that we start extrapolating by widening only after some fixed number of iterations of a loop are analyzed explicitly. This way, we collect sufficient preconditions for ensuring definite termination in the form of lifted decision trees at all program locations. From the decision tree in the initial location, we can thus establish for any variant from which initial states it will definitely terminate.

We establish correctness of the lifted analysis based on $\mathbb{T}^L(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$ by showing that it produces identical results with the brute-force approach that analyzes all variants one-by-one using the domain $\mathbb{T}^T$. Let $[\![s]\!]_{\mathbb{T}^L}$ denote the transfer function of statement s in $\mathbb{T}^L(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$, while $[\![s]\!]_{\mathbb{T}^T}$ denotes the transfer function of statement s in $\mathbb{T}^T$. Given $t \in \mathbb{T}^L(\mathbb{C}_{\mathbb{D}}, \mathbb{T}^T)$, we denote by $\pi_k(t) \in \mathbb{T}^T$ the leaf node of tree t that corresponds to the variant $k \in \mathcal{K}$.

Theorem 4.1 (Correctness)**.**
$\pi_k([\![s]\!]_{\mathbb{T}^L}(t)) = [\![P_k(s)]\!]_{\mathbb{T}^T}(\pi_k(t))$ *for all* $k \in \mathcal{K}$.

Proof. The proof is by induction on the structure of s. We consider the most interesting case of `#if` (θ) s `#endif`.

Assume $k \models \theta$. The leaf node $\pi_k(t)$ is in $\texttt{F-FILTER}_{\mathbb{T}^L}(t, \theta)$ but not in $\texttt{F-FILTER}_{\mathbb{T}^L}(t, \neg\theta)$. Thus, we have the result of $\pi_k([\![\texttt{\#if}\ (\theta)\ s\ \texttt{\#endif}]\!]_{\mathbb{T}^L}(t))$ is $[\![P_k(s)]\!]_{\mathbb{T}^T}(\pi_k(t))$ by IH. On the other hand, $P_k(\texttt{\#if}\ (\theta)\ s\texttt{\#endif}) = P_k(s)$ and the result of RHS is $[\![P_k(s)]\!]_{\mathbb{T}^T}(\pi_k(t))$.

Assume $k \not\models \theta$. The leaf node $\pi_k(t)$ is in $\texttt{F-FILTER}_{\mathbb{T}^L}(t, \neg\theta)$ but not in $\texttt{F-FILTER}_{\mathbb{T}^L}(t, \theta)$. Thus, we have that the result of $\pi_k([\![\texttt{\#if}\ (\theta)\ s\ \texttt{\#endif}]\!]_{\mathbb{T}^L}(t))$ is $\pi_k(t)$. On the other hand, $P_k(\texttt{\#if}\ (\theta)\ s\ \texttt{\#endif}) = \texttt{skip}$, and thus the result of RHS is $[\![\texttt{skip}]\!]_{\mathbb{T}^T}(\pi_k(t)) = \pi_k(t)$ □

Example 4.2. Consider the program family `cav2006.c` from Section 2. We perform (backward) lifted termination analysis parameterized by the *polyhedra* domain. In order to enforce convergence of the analysis, we apply the widening operator at the loop head. The invariant inferred by our analysis at program locations ④ and ① are shown in Figs. 5 and 2. By back-propagating the invariant at ④, denoted $t_④$, via `#if` directive at loc. ② we obtain the invariant at ①, denoted $t_①$. That is, $t_① = \text{B-ASSIGN}_{\mathbb{T}^L}(\text{F-FILTER}_{\mathbb{T}^L}(t_④, \texttt{A} < \texttt{B}), \texttt{y} = 50) \sqcup_{\mathbb{T}^L} \text{B-ASSIGN}_{\mathbb{T}^L}(\text{F-FILTER}_{\mathbb{T}^L}(t_④, \texttt{A} \geq \texttt{B}), \texttt{y} = 51)$. $\text{F-FILTER}_{\mathbb{T}^L}(t_④, \texttt{A} < \texttt{B})$ will return a decision tree with a root node $(\texttt{A} < \texttt{B})$ where only the left subtree of $t_④$ is kept, and after back-propagating $\texttt{y} = 50$ we will obtain the left subtree of $t_①$. Similarly, we obtain the right subtree of $t_①$.

5 Termination-Directed Program Sketching

In this section, we first introduce the language for writing program sketches and define their transformation to program families. Then, we extend the lifted decision tree domain to obtain more efficient program sketcher. Finally, we present an algorithm for solving termination-directed sketches.

5.1 Program Sketches

The language for sketches includes the same expression and statement productions as the language for program families, except that the `#if` statement is not allowed and a new

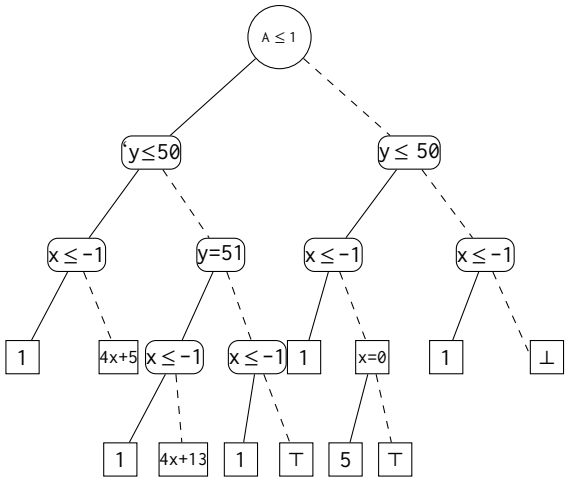

Figure 5. Lifted decision tree at loc. ④ of `cav2006.c`.

sketching construct is allowed represented by a basic numerical hole, ??. That is, $ae ::= \ldots \mid ??$. The numerical hole ?? is a placeholder that the synthesizer must replace with a suitable integer constant, such that the resulting program will always terminate under all possible inputs. Each hole occurrence in a program sketch $\hat{s}$ is uniquely labelled as $??_i$ and has a bounded integer domain $[n_i, n'_i]$. We will sometimes write $??_i^{[n_i,n'_i]}$ to make explicit the domain of a hole.

We want to transform an input program sketch $\hat{s}$ with a set of m holes $??_1^{[n_1,n'_1]}, \ldots, ??_m^{[n_m,n'_m]}$ into an output program family s with a set of numerical features $A_1, \ldots, A_m$ with domains $[n_1, n'_1], \ldots, [n_m, n'_m]$, respectively. The set of configurations $\mathcal{K}$ includes all possible combinations of feature's values. If a hole occurs in a linear expression that can be exactly represented in the numerical domain $\mathbb{D}$, then we can handle the hole in a more efficient *symbolic way* by an extended lifted termination 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: $\alpha_1 x_1 + \ldots \alpha_i ?? + \ldots \alpha_n x_n + \beta$, where $\alpha_1, \ldots, \alpha_n, \beta \in \mathbb{Z}$ and $x_1, \ldots x_n$ are program variables or other hole occurrences. Similarly, define when a hole can be exactly represented in the interval and octagon domains.

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

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

Otherwise, we use the *explicit rewrite rule*:

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

The set of features $\mathcal{F}$ is also updated with the fresh feature `A`. We write `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.

Note that in a program family `Rewrite`$(\hat{s})$ obtained using (SR) and (ER) rules feature variables can occur in arbitrary expressions, not only in `#if`-s as in traditional program families as defined in Section 3. Therefore, the syntax of arithmetic expressions in program families is now extended with feature variables, $ae ::= \ldots \mid A \in \mathcal{F}$. Moreover, the projection function $\mathtt{P}_k$ is updated accordingly to take into account feature variables. Thus, $\mathtt{P}_k$ recursively pre-processes all sub-statements and sub-expressions of statements. For example, $P_k(\mathtt{x:=}ae) = \mathtt{x:=}P_k(ae)$, $P_k(ae \oplus ae') = P_k(ae) \oplus P_k(ae')$, etc. Finally, P_k replaces a feature A with the value $k(A) \in \mathbb{Z}$, that is $P_k(A) = k(A)$.

Example 5.1. Reconsider the sketch `loop1.c` in Fig. 3. Both holes ?? can be represented exactly in the polyhedra domain P, so we use (SR) rule to obtain the corresponding program family. In contrast, consider the sketch `vmcai2004a.c` from SV-COMP: `int x; while (x ≥ 0) x = ??*x+10`. The hole ?? occurs in a non-linear expression `??*x+10`, so we use (ER) rule to obtain the corresponding program family. □

Let H be a set of holes in a program sketch. We define a *control function* $\phi : H \to \mathbb{Z}$ to describe the value of each hole in a sketch. We denote by $\hat{s}^{\phi}$ a candidate solution to the sketch $\hat{s}$ fully described by the control function ϕ. Let $[[s]]$ denotes the standard semantics of a single-program s [35]. The following result can be proved by structural induction on statements and expressions.

Theorem 5.2. *Let $\hat{s}$ be a sketch with holes $??_1, \ldots, ??_n$, and ϕ be a control function. Let $\overline{s}$ = `Rewrite`$(\hat{s})$ be a program family, in which features $A_1, \ldots, A_n$ correspond to holes $??_1, \ldots, ??_n$. We define a configuration $k \in \mathcal{K}$, s.t. $k(A_i) = \phi(??_i)$ for $1 \le i \le n$. Then, we have: $[[\hat{s}^{\phi}]] = [[\mathtt{P}_k(\overline{s})]]$.*

5.2 Extended Lifted Termination Analysis

Feature variables obtained by (SR) rule can freely occur in arbitrary expressions in the program family `Rewrite`$(\hat{s})$, not only in presence conditions of `#if`-s as in traditional program families (see Section 3). Hence, assignments `x:=`ae and tests be in `Rewrite`$(\hat{s})$, where ae and be may contain both program and feature variables from $Var \cup \mathcal{F}$, might also impact some linear constraints within decision nodes as well as some ranking functions within leaf nodes. Therefore, we define extended (improved) versions of B-ASSIGN$_{\mathbb{T}^L}$ and FILTER$_{\mathbb{T}^L}$ that take into account possibility of features occurring in expressions, and so they can modify both leaf and decision nodes. The lifted decision tree domain $\mathbb{T}^L(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{T}^T)$ is now slightly refined, such that domain $\mathbb{T}^T$ is defined over both program and feature variables, $Var \cup \mathcal{F}$.

B-ASSIGN$_{\mathbb{T}^L}(t, \mathtt{x:=}ae, C)$ (resp., FILTER$_{\mathbb{T}^L}(t, be, C)$), given in Algorithm 4 (resp., Algorithm 5), accumulates into the

Algorithm 4: $\texttt{B-ASSIGN}_{\mathbb{T}^L}(t, \texttt{x:=}ae, C)$

1 **if** isLeaf(t) **then**
2 $\quad t' = \texttt{B-ASSIGN}_{\mathbb{T}^T}(t, \texttt{x:=}ae)$;
3 $\quad$ **return** $\ll\texttt{FILTER}_{\mathbb{T}^T}(t', C)\gg$
4 **if** isNode(t) **then**
5 $\quad l = \text{B-ASSIGN}_{\mathbb{T}}(t.l, \texttt{x:=}ae, C \cup \{t.c\})$;
6 $\quad r = \text{B-ASSIGN}_{\mathbb{T}}(t.r, \texttt{x:=}ae, C \cup \{\neg t.c\})$;
7 $\quad$ **return** $[[t.c : l, c]]$

Algorithm 5: $\texttt{FILTER}_{\mathbb{T}^L}(t, be, C)$

1 **if** isLeaf(t) **then**
2 $\quad t' = \texttt{FILTER}_{\mathbb{T}^T}(t \uplus \alpha_{\mathbb{C}_\mathbb{D}}(C), be)$;
3 $\quad J = \gamma_{\mathbb{C}_\mathbb{D}}(t' \restriction_{\mathcal{F}})$;
4 $\quad$ **if** isRedundant(J, C) **then return** $\ll t'\gg$;
5 $\quad$ **else return** $\texttt{RESTRICT}(\ll t'\gg, C, J \backslash C)$;
6 **if** isNode(t) **then**
7 $\quad l = \text{FILTER}_{\mathbb{T}^L}(t.l, be, C \cup \{t.c\})$;
8 $\quad r = \text{FILTER}_{\mathbb{T}^L}(t.r, be, C \cup \{\neg t.c\})$;
9 $\quad$ **return** $[[t.c : l, r]]$

set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}_\mathcal{F}})$ (initialized to $\mathcal{K}$), constraints encountered along the paths of the decision tree (Lines 5,6) (resp., (Lines 7,8)), up to the leaf nodes where assignment $\texttt{x:=}ae$ is handled by $\text{B-ASSIGN}_{\mathbb{T}^T}$ (resp., test be is handled by $\text{FILTER}_{\mathbb{T}^T}$ applied on an element obtained by merging $\uplus$ constraints from the leaf node and the set C) (Line 2). The obtained result t' in $\text{B-ASSIGN}_{\mathbb{T}^L}$ is restricted to satisfy the accumulated constraints C by using $\text{FILTER}_{\mathbb{T}^T}$ (Line 3), whereas the result t' in $\text{FILTER}_{\mathbb{T}^L}$ is projected on features by $\restriction_{\mathcal{F}}$ to generate a new set of constraints J that are added to the given path using the function RESTRICT (Line 5).

5.3 Solving Sketches

We can now reduce the termination-directed sketching problem to an lifted termination analysis problem. Once the lifted termination analysis of the corresponding program family is performed, we can see from the inferred lifted decision trees in the initial location for which variants the *fully-defined* ranking function is generated. Those variants that satisfy the encountered linear constraints along the valid top-down paths, represent the correct sketch realizations.

The synthesis algorithm for solving a sketch $\hat{s}$ is described by Algorithm 6. First, a program sketch $\hat{s}$ is transformed into a program family $\overline{s} = \texttt{Rewrite}(\hat{s})$ (Line 1). Second, the extended lifted termination analysis of $\overline{s}$ is performed, which takes as input the lifted decision tree t_{in} at the final location and the program family $\overline{s}$ and returns a lifted decision tree in the initial location (Line 2). Finally, the returned lifted decision tree t is analyzed using the function FindDefined (Line 3). It reports a set of variants $\mathcal{K}' \subseteq \mathcal{K}$ for which ranking functions (leaf nodes) *fully-defined* over all possible values of input variables are found. The following result follows from the correctness of Rewrite (see Theorem 5.2) and the correctness of lifted termination analysis (see Theorem 4.1).

Theorem 5.3. SolveSketch*($\hat{s}$) is correct and terminates.*

Algorithm 6: $\texttt{SolveSketch}(\hat{s} : Stm)$

1 $\overline{s} = \texttt{Rewrite}(\hat{s})$;
2 $t = [[\overline{s}]]_{\mathbb{T}^L} t_{in}$;
3 **return** $\texttt{FindDefined}(t)$;

6 Evaluation

We evaluate our decision tree-based approach for lifted termination analysis and termination-directed program sketching by comparing its performances against the tuple-based lifted analysis and the variability encoding approach. All above approaches are based on the FuncTion tool [34–36], which represents a single-program termination analyzer phrased in the abstract interpretation framework. A detailed comparison of the FuncTion with other single-program termination analyzers [5, 19] can be found in [34–36]. To the best of our knowledge, this is the first study of lifted termination analysis and termination-directed program sketching.

Lifted tuple-based termination analysis. The lifted tuple-based domain for representing lifted ranking functions is $\langle \prod_{k\in\mathcal{K}} \mathbb{T}^T, \dot{\sqsubseteq}\rangle$. That is, there is one separate copy of $\mathbb{T}^T$ for each configuration $k \in \mathcal{K}$. Hence, the elements of the lifted domain are tuples $\overline{t'}$ that maintain one property element (a piecewise-defined ranking function from $\mathbb{T}^T$) per configuration. All lifted operations and transfer functions are defined by lifting the corresponding operations and transfer functions of the domain $\mathbb{T}^T$ configuration-wise. For example, the ordering is defined as: $\overline{t'_1}\dot{\sqsubseteq}\overline{t'_2} \equiv \pi_k(\overline{t'_1}) \sqsubseteq_{\mathbb{T}^T} \pi_k(\overline{t'_2})$, for $\forall k \in \mathcal{K}$, where the projection π_k selects the k^{th} component of tuple $\overline{t'}$. We refer to [15] for similar definitions of lifted operations and transfer functions of the lifted tuple-based domain $\langle \prod_{k\in\mathcal{K}} \mathbb{D}_{Var}, \dot{\sqsubseteq}\rangle$. Note that the tuple-based analysis represents a simple lifted (variability-aware) analysis [4, 27], since it works on the level of program families. This approach improves over the "brute force" strategy since a lot of caching effects and improvements are possible for it. For example, it compiles and executes the fixed point iterative algorithm once per whole family, configuration satisfiability tests $k \models \theta$ can be memoized, many transfer functions that act identically for all configurations can be executed only once, etc. Therefore, as evidenced in [4, 27], the lifted tuple-based analysis is faster than the "brute force" approach.

Implementation. We have developed a prototype lifted termination analyzer, called SPLFuncTion, which uses lifted domains of tuples $\prod_{k\in\mathcal{K}} \mathbb{T}^T$ and decision trees $\mathbb{T}^L$. The tool

also supports synthesis algorithm based on tuples and decision trees for solving termination-directed program sketches. It is built on top of the FuncTion tool [34] for inferring piecewise-defined ranking functions of single programs. The well-known numerical abstract domains, such as intervals, octagons, and polyhedra, are used as parameters in our lifted domains. Their abstract operations and transfer functions are provided by the APRON library [21]. Our proof-of-concept implementation is written in OCaml and consists of around 7K LOC. The tool accepts programs written in a subset of C with `#if` directives. It currently provides only a limited support for arrays, pointers, `struct` and `union` types. The only basic data type is mathematical integers.

Experiment setup and Benchmarks. 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 200 seconds. All times are reported as average over five independent executions. The implementation, benchmarks, and all obtained results are available from: https://github.com/aleksdimovski/SPLFunction. In our experiments, we use three instances of our lifted termination analysis via decision trees: $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$; and three instances of lifted termination analysis via tuples: $\overline{\mathcal{A}}_{\Pi}(I)$, $\overline{\mathcal{A}}_{\Pi}(O)$, and $\overline{\mathcal{A}}_{\Pi}(P)$, which use intervals, octagons, and polyhedra domains as parameters, respectively. For each version of decision tree-based lifted analysis, we report the running time in seconds (denoted by Time) to analyze the given benchmark, and we report the speed up factor (denoted by Impr.) for each lifted analysis based on decision trees relative to the corresponding lifted analysis based on tuples ($\overline{\mathcal{A}}_{\mathbb{T}}(-)$ vs. $\overline{\mathcal{A}}_{\Pi}(-)$). Additionally, we consider the single-program termination analysis $\mathcal{A}(P)$, as implemented by the FuncTion tool [34], applied to variability simulators of given program families.

The evaluation is performed on a dozen of C loop programs collected from several categories of SV-COMP suite: `termination-crafted-lit` (written `crafted-lit` for short), `termination-crafted` (`crafted`), `termination-restrict` (`restricted`); from FuncTion project [34, 35]; as well as on several C numerical sketches collected from the `Sketch` project [30, 31]. In the case of SV-COMP, we have first selected some loop programs with integer variables, and then we have manually added variability (features and `#if`-s) in each of them. We have inserted presence conditions with different complexities, from atomic to more complex feature expressions, and `#if`-s are placed in different locations of the code. Tables 1 and 2 present characteristics of the benchmarks, such as: the file name (Benchmark), the category where it is located (folder), the number of features ($|\mathcal{F}|$), configurations ($|\mathcal{K}|$), and lines of code (LOC). Some of the benchmarks for termination analysis are given in Figs. 1 and 6 to 9, whereas the benchmarks for sketching are given in Figs. 3 and 10 to 13.

Performance Results. Table 1 shows the results of analyzing our benchmarks by using different versions of our lifted termination analyses based on decision trees and tuples. The performance results confirm that sharing is indeed effective and especially so for large values of $|\mathcal{K}|$. On our benchmarks, it translates to speed ups (i.e., ($\overline{\mathcal{A}}_{\mathbb{T}}(-)$ vs. $\overline{\mathcal{A}}_{\Pi}(-)$) that range from 1.6 to 38 times when $|\mathcal{K}| < 100$, and from 5.8 to 217 times when $|\mathcal{K}| > 100$. We can also notice that $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ is the fastest version, then it comes $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ is the slowest decision tree-based analysis but the most precise.

Consider `mccarthy91.c` code snippet given in Fig. 6, which represents an iterative implementation of the McCarthy 91 function [26]. Depending on the values assigned to feature `A` at compile-time, variable $\mathtt{x}_1$ is initialized to [0,100] when ($\mathtt{A} < 2$) and to [101,110] when ($\mathtt{A} \geq 2$). The lifted termination analysis infers the ranking function $[[\mathtt{A} \geq 2, 8, \top]]$, meaning that variants satisfying $\mathtt{A} \geq 2$ terminate in at most 8 execution steps, whereas for variants satisfying $\mathtt{A} < 2$ an indefinite (I don't know) answer is reported. In the case of `PastaB2.c` given in Fig. 7, SPLFuncTion establishes termination for variants satisfying ($\mathtt{Min} \leq \mathtt{A} \leq 2$) $\wedge$ ($\mathtt{Min} \leq \mathtt{B} \leq 2$). For example, the ranking function is: $3\mathtt{x} - 3\mathtt{y} + 7$ when ($\mathtt{x}-\mathtt{y} \geq 5$). Otherwise, an infinite behaviour is possible: when ($\mathtt{Min} \leq \mathtt{A} \leq 2$) $\wedge$ ($3 \leq \mathtt{B} \leq \mathtt{Max}$) a possible non-termination is reported if ($\mathtt{x} > \mathtt{y}$). For `sas2014b.c` given in Fig. 8, we prove termination for variants satisfying ($\mathtt{A} \leq \mathtt{B}+1$). For example, the ranking function is: $3\mathtt{x} - 2$ when ($\mathtt{x} \geq \mathtt{y} + 1$) $\wedge$ ($\mathtt{y} \geq 1$). Finally, SPLFuncTion proves termination for all variants of `boolean.c` in Fig. 9 by reporting ranking function: 303.

We have also applied the single-program termination analysis $\mathcal{A}(P)$ on variability simulators of program families in Table 1. For `mccarthy91.c` we obtain the $\top$ ('I don't know') answer; for `PastaB2.c` we obtain the $\bot$ answer when ($\mathtt{x} \geq \mathtt{y}$); while for `sas2014b.c` we obtain the $\bot$ answer when ($\mathtt{x} > 0 \vee \mathtt{y} > 0$). For all other benchmarks, we also obtain less precise ranking functions than using $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, except for `boolean.c` when the same answers are obtained. The lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ do not lose any precision with respect to feature variables, since domain elements are lifted decision trees that partition the space of all possible values of features inducing disjunctions into the base termination domain. Thus, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ produces identical (precision-wise) results with the brute-force approach (see Theorem 4.1). On the other hand, the variability encoding approach uses a single-program termination analysis $\mathcal{A}(P)$, where domain elements are decision trees that partition the space of possible values of feature and program variables inducing disjunctions into the base domain of affine functions. Still, this partitioning is more coarse: the join of neighbouring leaf nodes is often invoked for many operations thus producing undefined results as evidenced by the above experiments.

Table 2 shows the results of synthesizing our sketches by using Polyhedra parameterized instances of our lifted

```
void main(){
  int x1, x2 = 1;
  #if (A<2) x1 = [0, 100];
  #else x1 = [101, 110];#endif
  while (x2 ≥ 1) {
    if {
      x1 = x1-10; x2 = x2-1;
    } else {
      x1 = x1+11; x2 = x2+1;}
  }
  return x1; }
```

Figure 6. mccarthy91.c.

```
void main(){
  int x;
  int y;
  while (x > y) {
    #if (A≤2) x = x-1;
    #else x = x+1; #endif
    #if (B≤3) y = y+1;
    #else y = y-1; #endif
  }
}
```

Figure 7. PastaB2.c.

```
void main(){
  int x;
  int y;
  while (x > 0 ∧ y > 0) {
    #if (A≤B+1) x = x-y;
    #else x = x+y; #endif
  }
}
```

Figure 8. sas2014b.c.

```
void main(){
 int x;
 #if (A>1)∧(A≤3) x=[-99, 0];
 #else x = [1, 99]; #endif
 while (x) {
 if (x>0) x = x-1;
else x = x+1;
 }
}
```

Figure 9. boolean.c.

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

Figure 10. tap2008a.c.

```
void main(){
  int x=??;
  while (x > 0 ∧ x < 50) {
    if (x<20) x = x-1;
    if (x > 10) x=x+1;
    if (30 ≤ x ∧ x ≤ 40) x=x-1;
  }
}
```

Figure 11. tap2008e.c.

```
void main(){
  int x;
  int K=??;
  while (x ≠ K) {
    if (x>K) x = x-1;
    else x = x+1;
  }
}
```

Figure 12. tacas2013b.c.

```
void main() {
  int x = ??1;
  while (x  ≥ 0) {
    if (y <??2) x = x-2;
  }
}
```

Figure 13. loop2.c.

Table 1. Performance results for lifted termination analyses based on decision trees $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ vs. tuples $\overline{\mathcal{A}}_{\Pi}(-)$ (which are used as baseline). All times are in seconds.

Benchmark	folder	LOC	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{K}\rvert$	$\overline{\mathcal{A}}_{\mathbb{T}}(I)$		$\overline{\mathcal{A}}_{\mathbb{T}}(O)$		$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	
					Time	Impr.	Time	Impr.	Time	Impr.
cav2006.c	crafted-lit	20	2	8	0.128	2.5×	0.168	1.6×	0.162	3×
mccarthy91.c	crafted	25	1	32	0.097	35×	0.111	35×	0.269	38×
PastaB2.c	restrict	17	2	128	0.001	13×	0.031	5.8×	0.039	9.6×
sas2014b.c	crafted	15	2	256	0.005	40×	0.015	45×	0.038	95×
boolean.c	FuncTion	20	1	16	0.011	12×	0.010	10.9×	0.020	12×
issue8.c	FuncTion	20	1	8	0.008	5.5×	0.019	6.6×	0.027	7.1×
example5.c	FuncTion	20	2	256	0.006	183×	0.007	125×	0.009	190×
Mysore.c	crafted	15	1	16	0.001	5×	0.002	6.5×	0.020	10×
Copenhagen.c	crafted	20	3	512	0.082	150×	0.125	92×	0.100	217×

analyses $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ and $\overline{\mathcal{A}}_{\Pi}(P)$. We consider various versions of our benchmarks with 4, 5, and 8-bit sizes of holes. Note that a sketch with n-bit holes is translated into a family with features that have 2^n possible values. For example, loop1.c sketch with two 5-bit holes will be transformed into a program family with $2^5 \cdot 2^5 = 1024$ configurations. The decision tree-based approach scales better for all benchmarks giving speed ups that range from 13.8 to 78 times for 4-bit sketches, and from 28.3 to 406 times for 5-bit sketches. In the case of 8-bit sketches, $\overline{\mathcal{A}}_{\Pi}(P)$ often times out due to the large configuration spaces (e.g. $\lvert\mathcal{K}\rvert = 65536$ for sketches with two holes). All sketches with holes that can be handled symbolically by (SR) rule can be efficiently analyzed using $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, which does not depend on the sizes of holes in those cases.

Consider the sketch tap2008a.c given in Fig. 10. It contains one hole ?? (corresponding feature A) that can be handled symbolically by (SR) rule. Hence, our decision tree-based synthesis approach has similar running times for all domain sizes, reporting as solutions the constraints: $(\texttt{Min} \le \texttt{A} \le 24) \wedge (31 \le \texttt{A} \le \texttt{Max})$. That is, the sketch tap2008a.c always terminates when the hole ?? is either less than 25 or larger than 30. The obtained ranking function is: 3 when $(\texttt{A} \le 10)$; $4\texttt{A}-37$ when $(10 < \texttt{A} \le 24)$, and 47 when $(\texttt{A} \ge 31)$. Similarly,

Table 2. Performance results for termination-directed sketching based on decision trees $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ vs. tuples $\overline{\mathcal{A}}_{\Pi}(P)$ (which is used as baseline). All times in seconds.

Benchmarks	folder	LOC	$\lvert\mathcal{F}\rvert$	4 bits		5 bits		8 bits	
				Time	Impr.	Time	Impr.	Time	Impr.
loop1.c	Sketch	20	2	0.074	74×	0.074	367×	0.074	timeout
tap2008a.c	FuncTion	25	1	0.040	15×	0.040	30×	0.041	374×
tap2008e.c	FuncTion	25	1	0.267	17.7×	0.269	41.5×	0.276	496×
tacas2013b.c	crafted	20	1	0.027	13.8×	0.027	28.3×	0.027	281×
loop2.c	Sketch	20	2	0.058	78×	0.058	406×	0.060	timeout

SPLFuncTion establishes that tap2008e.c given in Fig. 11 terminates for constraints: $(\texttt{Min} \leq \texttt{A} \leq 11) \wedge (40 \leq \texttt{A} \leq \texttt{Max})$. The obtained ranking function is: $5\texttt{A} + 3$ when $(\texttt{A} \leq 11)$; $-5\texttt{A} + 253$ when $(40 \leq \texttt{A} \leq 49)$, and 3 when $(\texttt{A} \geq 50)$. We obtain that tacas2013b.c sketch terminates for any value for the hole ?? (corresponding feature A). The ranking function is: $3\texttt{A} - 3\texttt{x} + 3$ when $(\texttt{A-x} \geq 0)$; and $-3\texttt{A} + 3\texttt{x} + 3$ when $(\texttt{A-x} \leq 0)$. Finally, the sketcher synthesizes a solution for loop2.c that satisfies $(??_1 \leq ??_2 - 1)$. For example, the ranking function is: $3??_1 + 11$ when $(??_1 \geq 4)$.

Threats to validity. Our current tool supports a non-trivial subset of C, and the missing constructs (e.g. pointers, struct and union types) are largely orthogonal to the solution (lifted decision tree domain). That is, supporting these constructs would not provide any new insights to our evaluation. We perform lifted termination analysis of relatively small benchmarks. However, the focus of the lifted decision tree domain is to combat the configuration space blow-up of program families, not their LOC size. So, we expect to obtain similar or better results for larger benchmarks. Another threat to validity is the synthetic variability that has been manually added to the benchmarks. We have generated benchmarks with various configuration spaces, ranging from 16 to 65536 configurations. We have also inserted presence conditions with different complexities, from atomic to more complex feature expressions, and #if-s are placed in different code locations (e.g., initialization, inside loops, etc).

7 Related Work

Research on behavioral analysis of program families has been a hot topic in the past decade. Various static analysis techniques have been successfully lifted to work at the level of program families. They can be classified based on the underlying technologies that are used: abstract interpretation [11, 13, 15, 17, 27], data-flow analysis [3, 4, 37], type checking [23, 24], type-based analysis [7], model checking [2, 10, 12, 18], theorem proving [33] etc. Our approach belongs to the abstract interpretation category. Midtgaard et. al. [27] have proposed the lifted tuple-based analysis phrased in the abstract interpretation framework [8, 29], while the works [11, 13] improve the tuple representation by using lifted binary decision diagram (BDD) domains. They are applied to classical program families with only Boolean features. Subsequently, the lifted decision tree domain [15] has been proposed to handle program families with both Boolean and numerical features, which represent the majority of industrial embedded code. Moreover, this domain has been extended to analyze dynamic program families [14]. However, the above lifted analyses are forward and infer numerical invariants, whereas here we consider a backward analysis inferring ranking functions.

In the recent past, several powerful termination provers have been constructed. For example, the Terminator prover [5] is based on iterative construction of transition invariants, the FuncTion tool [35] is based on abstract interpretation, while the tool in [19] represents a constraint-based approach.

The Sketch tool [30, 31] uses SAT-based inductive synthesis to resolve the assertion-directed program sketches. It reasons about loops by unrolling them, so is very sensitive to the degree of unrolling. Our approach being based on abstract interpretation uses widening instead of fully unrolling loops, so that we can handle directly unbounded loops in a sound way. Still, our approach is based on numerical abstract domains, so it can be only applied for synthesizing numerical programs. The Sketch tool is more general and especially successful for bit-manipulating programs.

Recently, some researchers have explored ways to reduce the sketching synthesis problem as a lifted analysis problem. Ceska et. al. [6] use a counterexample guided abstraction refinement technique for analyzing product lines to resolve probabilistic PRISM sketches. In a similar vein as here, the work [16] uses a lifted numerical analysis based on abstract interpretation to resolve assertion-directed sketches.

8 Conclusion

In this work, we employ techniques from abstract interpretation and product-line analysis for performing termination analysis of program families and for automatic resolving of program sketches. By means of an implementation and a number of experiments, we show that our approach is effective and performs well on a variety of C benchmarks.

References

[1] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer.

[2] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).* 372–375. https://doi.org/10.1109/ASE.2011.6100075

[3] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL$^{\text{LIFT}}$: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13.* 355–364.

[4] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. 2013. Intraprocedural Dataflow Analysis for Software Product Lines. *T. Aspect-Oriented Software Development* 10 (2013), 73–108.

[5] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better Termination Proving through Cooperation. In *Computer Aided Verification - 25th International Conference, CAV 2013. Proceedings (LNCS, Vol. 8044).* Springer, 413–429. https://doi.org/10.1007/978-3-642-39799-8_28

[6] Milan Ceska, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. 2019. 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).* Springer, 107–125. https://doi.org/10.1007/978-3-030-31514-6_7

[7] Sheng Chen and Martin Erwig. 2014. Type-based parametric analysis of program families. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming.* 39–51.

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Record of the Fourth ACM Symposium on POPL.* ACM, 238–252. https://doi.org/10.1145/512950.512973

[9] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on POPL'78.* ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[10] Aleksandar S. Dimovski. 2018. Verifying annotated program families using symbolic game semantics. *Theor. Comput. Sci.* 706 (2018), 35–53. https://doi.org/10.1016/j.tcs.2017.09.029

[11] Aleksandar S. Dimovski. 2019. Lifted static analysis using a binary decision diagram abstract domain. In *Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019.* ACM, 102–114. https://doi.org/10.1145/3357765.3359518

[12] Aleksandar S. Dimovski. 2020. CTL* family-based model checking using variability abstractions and modal transition systems. *Int. J. Softw. Tools Technol. Transf.* 22, 1 (2020), 35–55. https://doi.org/10.1007/s10009-019-00528-0

[13] Aleksandar S. Dimovski. 2021. A binary decision diagram lifted domain for analyzing program families. *Journal of Computer Languages* 63 (2021), 101032. https://doi.org/10.1016/j.cola.2021.101032

[14] Aleksandar S. Dimovski and Sven Apel. 2021. Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation. In *35th European Conference on Object-Oriented Programming, ECOOP 2021 (LIPIcs, Vol. 194).* Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2021.247

[15] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2021. 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).* Springer, 67–86. https://arxiv.org/abs/2012.05863

[16] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2021. Program Sketching using Lifted Analysis for Numerical Program Families. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings (LNCS, Vol. 12673).* Springer, 95–112. https://doi.org/10.1007/978-3-030-76384-8_7

[17] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2018. Variability abstractions for lifted analysis. *Sci. Comput. Program.* 159 (2018), 1–27.

[18] Aleksandar S. Dimovski, Axel Legay, and Andrzej Wasowski. 2020. Generalized abstraction-refinement for game-based CTL lifted model checking. *Theor. Comput. Sci.* 837 (2020), 181–206. https://doi.org/10.1016/j.tcs.2020.06.011

[19] Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. 2013. Linear Ranking for Linear Lasso Programs. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013. Proceedings (LNCS, Vol. 8172).* Springer, 365–380. https://doi.org/10.1007/978-3-319-02444-8_26

[20] Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. 2015. Experiences from Designing and Validating a Software Modernization Transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015.* 597–607. https://doi.org/10.1109/ASE.2015.84

[21] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings (LNCS, Vol. 5643).* Springer, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

[22] Christian Kästner. 2010. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* Ph.D. Dissertation. University of Magdeburg, Germany.

[23] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.* 21, 3 (2012), 14.

[24] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: toward type checking #ifdef variability in C. In *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010, Eindhoven, Netherlands, October 10, 2010*, Sven Apel, Don S. Batory, Krzysztof Czarnecki, Florian Heidenreich, Christian Kästner, and Oscar Nierstrasz (Eds.). ACM, 25–32. https://doi.org/10.1145/1868688.1868693

[25] Tasnuva Mahjabin, Yang Xiao, Guang Sun, and Wangdong Jiang. 2017. A survey of distributed denial-of-service attack, prevention, and mitigation techniques. *Int. J. Distributed Sens. Networks* 13, 12 (2017). https://doi.org/10.1177/1550147717741463

[26] Zohar Manna. 1974. *Mathematical Theory of Computation.* McGraw-Hill.

[27] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2015. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.* 105 (2015), 145–170. https://doi.org/10.1016/j.scico.2015.04.005

[28] Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1

[29] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372. https://doi.org/10.1561/2500000034

[30] Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[31] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation.* ACM, 281–294. https://doi.org/10.1145/1065010.1065045

[32] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6.

[33] Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. 2012. Family-based deductive verification of software product lines. In *Generative Programming and Component Engineering, GPCE'12*. ACM, 11–20. https://doi.org/10.1145/2371401.2371404

[34] Caterina Urban. 2015. 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 (LNCS, Vol. 9035)*. Springer, 464–466. https://doi.org/10.1007/978-3-662-46681-0_46

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

[36] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Static Analysis - 21st International Symposium, SAS 2014. Proceedings (LNCS, Vol. 8723)*. Springer, 302–318. https://doi.org/10.1007/978-3-319-10936-7_19

[37] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33. https://doi.org/10.1145/3280986

[38] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *J. Log. Algebraic Methods Program.* 85, 1 (2016), 125–145. https://doi.org/10.1016/j.jlamp.2015.06.007