A Decision Tree Abstract Domain for Analyzing Program Families with Numerical Features

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

[1] Mother Teresa University, 12 Udarna Brigada 2a, 1000 Skopje, N. Macedonia
[2] Saarland University, Campus E1.1, 66123 Saarbrücken, Germany
[3] Université catholique de Louvain, 1348 Ottignies-Louvain-la-Neuve, Belgium

Abstract. *Lifted* (*family-based*) *static analysis* by abstract interpretation is capable of analyzing all variants of a program family simultaneously, in a single run. The elements of the underlying lifted analysis domain are tuples, which maintain one property per variant. Still, explicit property enumeration in tuples, one by one for all variants, immediately yields combinatorial explosion. This is particulary apparent in the case of program families that, apart from Boolean features, contain also numerical features with big domains, thus admitting astronomic configuration spaces.
In this work, we introduce a new symbolic representation of the lifted abstract domain that can efficiently analyze program families with numerical features. This makes sharing between property elements corresponding to different variants explicitly possible. The elements of the new lifted domain are constraint-based *decision trees*, where decision nodes are labeled with linear constraints defined over numerical features and the leaf nodes belong to an existing single-program analysis domain. To illustrate the potential of this representation, we have implemented an experimental lifted static analyzer, called SPLNum^{2}Analyzer, for inferring invariants of C programs. It uses existing numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library as parameters. An empirical evaluation on benchmarks from SV-COMP and BusyBox yields promising preliminary results indicating that our decision trees-based approach is effective and significantly outperforms the tuple-based approach, which is used as a baseline analysis based on abstract interpretation.

1 Introduction

Many software systems today are configurable [7]: they use *features* (or configurable options) to control presence and absence of software functionality. Different family members, called variants, are derived by switching features on and off, while reuse of common code is maximized, leading to productivity gains, shorter time to market, greater market coverage, etc. Program families (e.g., Software Product Lines) are commonly seen in the development of commercial embedded software, such as cars, phones, avionics, medicine, robotics, etc. We consider here program families implemented using `#if` directives from the C preprocessor `CPP` [18]. They use `#if`-s to specify under which conditions parts of code should be included or excluded from a variant. Classical program families use only

Boolean features that have two values: on and off. However, Boolean features are insufficient for real-world program families, as there exist features that have a range of numbers as possible values. These features are called *numerical features* [16,22]. For instance, Linux kernel, BusyBox, Apache web server, Java Garbage Collector represent some real-world program families with numerical features. Analyzing such program families is very challenging, due to the fact that from only a few features, huge number of variants can be derived.

This paper concerns the verification of program families with Boolean and numerical features using abstract interpretation-based static analysis. *Abstract interpretation* [8,21] is a general theory for approximating the semantics of programs. It provides sound (all answers are correct) and efficient (with a good trade-off between precision and cost) static analyses of run-time properties of real programs. Still, static analysis of program families is harder than static analysis of single programs, because the number of possible variants can be very large (often huge) in practice. The simplest brute-force approach that 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, is very inefficient [4]. Therefore, we use so-called *lifted* (family-based) *static analyses* [4,19,24], which analyze all variants of the family simultaneously. They take as input the common code base, which encodes all variants of a program family, and produce precise analysis results corresponding to all variants. They use a lifted analysis domain, which represents a n-fold product of an existing single-program analysis domain used for expressing program properties (where n is the number of valid configurations). That is, the lifted analysis domain maintains one property element per valid variant in tuples. The problem is that this explicit property enumeration in tuples becomes computationally intractable with larger program families because the number of variants grows exponentially with the number of features. This problem has been successfully addressed for program families that contain only Boolean features [1,2,3,13], by using sharing through binary decision diagrams (BDDs). However, the fundamental limitation of existing lifted analysis techniques is that they do not deal with numerical features.

To overcome this limitation, in this work we present a new, refined lifted abstract domain for effectively analyzing program families with numerical features by means of abstract interpretation. The elements of the lifted abstract domain are constraint-based *decision trees*, where the decision nodes are labelled with linear constraints over numerical features, whereas the leaf nodes belong to a single-program analysis domain. The decision trees recursively partition the space of configurations (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 the variants (configurations) that satisfy the constraints along the path to the given leaf node. The partitioning is dynamic, which means that partitions are split by feature-based tests (at `#if` directives), and joined when merging the corresponding control flows again. In terms of decision trees, this means that new decision nodes are added by feature-based tests and removed when merging control flows. In fact, the partitioning of the set of configurations is semantics-based, which means that linear constraints over numerical features that occur in decision nodes are automatically inferred by the analysis and do not necessarily occur syntactically in the code base.

The lifted abstract domain is parametric in the choice of numerical property domain which underlies the linear constraints over numerical features labelling decision nodes, and the choice of the single-program analysis domain for leaf nodes. In fact, in our implementation, we also use numerical property domains for leaf nodes, which encode linear constraints over program variables. We use here the well-known numerical domains, such as intervals [8], octagons [20], polyhedra [12], from the APRON library [17] to obtain a concrete decision tree-based implementation of the lifted abstract domain. This way, we have implemented a *forward reachability analysis* of C program families with numerical (and Boolean) features for the automatic inference of invariants. Our tool, called SPLNum^{2}Analyzer[4], computes a set of possible invariants, which represent linear constraints over program variables. We can use the implemented lifted static analyzer to check invariance properties of C program families, such as assertions, buffer overflows, null pointer references, division by zero, etc [10].

2 Motivating Example

To illustrate the potential of a decision tree-based lifted domain, we consider a motivating example using the code base of the following program family SIMPLE:

```
①    int x := 10, y := 0;
②    while (x != 0) {
③        x := x-1;
④        #if (SIZE ≤ 3) y := y+1; #else y := y-1; #endif
⑤        #if (!B) y := 0;  #else skip; #endif ⑥}
⑦    assert (y > 1);
```

The set $\mathbb{F}$ of features is $\{\texttt{B}, \texttt{SIZE}\}$, where B is a Boolean feature and SIZE is a numerical feature whose domain is $[1, 4] = \{1, 2, 3, 4\}$. Thus, the set of valid configurations is $\mathbb{K} = \{\texttt{B} \wedge (\texttt{SIZE}=1), \texttt{B} \wedge (\texttt{SIZE}=2), \texttt{B} \wedge (\texttt{SIZE}=3), \texttt{B} \wedge (\texttt{SIZE}=4), \neg\texttt{B} \wedge (\texttt{SIZE} = 1), \neg\texttt{B} \wedge (\texttt{SIZE} = 2), \neg\texttt{B} \wedge (\texttt{SIZE} = 3), \neg\texttt{B} \wedge (\texttt{SIZE} = 4)\}$. The code of SIMPLE contains two #if directives, which change the value assigned to y, depending on how features from $\mathbb{F}$ are set at compile-time. For each configuration from $\mathbb{K}$, a different variant (single program) can be generated by appropriately resolving #if-s. For example, the variant corresponding to configuration $\texttt{B} \wedge (\texttt{SIZE}=1)$ will have B and SIZE set to true and 1, so that the assignment y := y+1 and skip in program locations ④ and ⑤, respectively, will be included in this variant. The variant for configuration $\neg\texttt{B} \wedge (\texttt{SIZE}=4)$ will have features B and SIZE set to false and 4, so the assignments y := y-1 and y := 0 in program locations ④ and ⑤, respectively, will be included in this variant. There are $|\mathbb{K}| = 8$ variants that can be derived from the family SIMPLE.

[4] Num^2 in the name of the tool refers to its ability to both handle NUMerical features and to perform NUMerical client analysis of SPLs (program families).

$$\Big(\overbrace{[\mathtt{y}=10,\mathtt{x}=0]}^{\mathtt{B}\wedge(\mathtt{SIZE}=1)}, \overbrace{[\mathtt{y}=10,\mathtt{x}=0]}^{\mathtt{B}\wedge(\mathtt{SIZE}=2)}, \overbrace{[\mathtt{y}=10,\mathtt{x}=0]}^{\mathtt{B}\wedge(\mathtt{SIZE}=3)}, \overbrace{[\mathtt{y}=-10,\mathtt{x}=0]}^{\mathtt{B}\wedge(\mathtt{SIZE}=4)}, \overbrace{[\mathtt{y}=0,\mathtt{x}=0]}^{\neg\mathtt{B}\wedge(\mathtt{SIZE}=1)}, \overbrace{[\mathtt{y}=0,\mathtt{x}=0]}^{\neg\mathtt{B}\wedge(\mathtt{SIZE}=2)}, \overbrace{[\mathtt{y}=0,\mathtt{x}=0]}^{\neg\mathtt{B}\wedge(\mathtt{SIZE}=3)}, \overbrace{[\mathtt{y}=0,\mathtt{x}=0]}^{\neg\mathtt{B}\wedge(\mathtt{SIZE}=4)}\Big)$$

Fig. 1: Tuple-based analysis result at location ⑦ of SIMPLE.

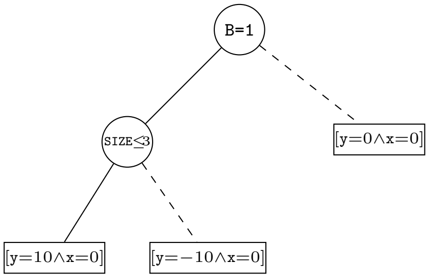

Fig. 2: Decision tree-based analysis result at location ⑦ of SIMPLE (solid edges = true, dashed edges = false).

Assume that we want to perform *lifted polyhedra analysis* of SIMPLE using the *Polyhedra* numerical domain [12]. The standard lifted analysis domain used in the literature [4,19] is defined as cartesian product of $|\mathbb{K}|$ copies of the basic analysis domain (e.g. polyhedra). Hence, elements of the lifted domain are tuples containing one component for each valid configuration from $\mathbb{K}$, where each component represents a polyhedra linear constraint over program variables (`x` and `y` in this case). The lifted analysis result in location ⑦ of SIMPLE is an 8-sized tuple shown in Fig. 1. Note that the first component of the tuple in Fig. 1 corresponds to configuration $\mathtt{B} \wedge (\mathtt{SIZE}=1)$, the second to $\mathtt{B} \wedge (\mathtt{SIZE}=2)$, the third to $\mathtt{B} \wedge (\mathtt{SIZE}=3)$, and so on. We can see in Fig. 1 that the polyhedra analysis discovers very precise results for the variable `y`: $(\mathtt{y}=10)$ for configurations $\mathtt{B} \wedge (\mathtt{SIZE}=1)$, $\mathtt{B} \wedge (\mathtt{SIZE}=2)$, and $\mathtt{B} \wedge (\mathtt{SIZE}=3)$; $(\mathtt{y}=-10)$ for configuration $\mathtt{B} \wedge (\mathtt{SIZE}=4)$; and $(\mathtt{y}=0)$ for all other configurations. This is due to the fact that the polyhedra domain is fully relational and is able to track all relations between program variables `x` and `y`. Using this result in location ⑦, we can successfully conclude that the assertion is valid for configurations $\mathtt{B} \wedge (\mathtt{SIZE}=1)$, $\mathtt{B} \wedge (\mathtt{SIZE}=2)$, and $\mathtt{B} \wedge (\mathtt{SIZE}=3)$, whereas the assertion fails for all other configurations.

If we perform lifted polyhedra analysis based on the *decision tree domain* proposed in this work, then the corresponding decision tree inferred in the final program location ⑦ of SIMPLE is depicted in Fig. 2. Notice that the inner nodes of the decision tree in Fig. 2 are labeled with *Interval* linear constraints over features (`SIZE` and `B`), while the leaves are labeled with the *Polyhedra* linear constraints over program variables `x` and `y`. Hence, we use two different numerical abstract domains in our decision trees: Interval domain [8] for expressing properties in decision nodes, and Polyhedra domain [12] for expressing properties in leaf nodes. 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 valid configurations $\mathbb{K}$, we implicitly assume the correctness of linear constraints that take into account domains of numerical features. For example, the node with constraint $(\mathtt{SIZE}\leq 3)$ is satisfied when $(\mathtt{SIZE}\leq 3) \land (1\leq \mathtt{SIZE}\leq 4)$, whereas its negation is satisfied when $(\mathtt{SIZE}>3) \land (1\leq \mathtt{SIZE}\leq 4)$. The constraints $(1\leq \mathtt{SIZE}\leq 4)$ represent the domain $[1,4]$ of $\mathtt{SIZE}$. We can see that decision trees offer more possibilities for sharing and interaction between analysis properties corresponding to different configurations, they provide symbolic and compact representation of lifted analysis elements. For example, Fig. 2 presents polyhedra properties of two program variables $\mathtt{x}$ and $\mathtt{y}$, which are partitioned with respect to features $\mathtt{B}$ and $\mathtt{SIZE}$. When $(\mathtt{B} \land (\mathtt{SIZE} \leq 3))$ is true the shared property is $(\mathtt{y} = 10, \mathtt{x} = 0)$, whereas when $(\mathtt{B} \land \neg(\mathtt{SIZE} \leq 3))$ is true the shared property is $(\mathtt{y} = -10, \mathtt{x} = 0)$. When $\neg\mathtt{B}$ is true, the property is independent from $\mathtt{SIZE}$, hence a node with a constraint over $\mathtt{SIZE}$ is not needed. Therefore, all such cases are identical and share the leaf $(\mathtt{y}=0, \mathtt{x}=0)$. In effect, the decision tree-based representation uses only three leafs, whereas the tuple-based representation uses eight. This ability for sharing is the key motivation behind the decision trees.

3 A Language for Program Families

Let $\mathbb{F} = \{A_1, \dots, A_k\}$ 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. Note that any Boolean feature can be represented as a numerical feature $B \in \mathbb{F}$ with $\mathrm{dom}(B) = \{0,1\}$, such that 0 means that feature B is disabled while 1 means that B is enabled. 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* $k : \mathbb{F} \to \mathbb{Z}$, which is a mapping that assigns a value from $\mathrm{dom}(A)$ to each feature A, i.e. $k(A) \in \mathrm{dom}(A)$ 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: $(A_1 = k(A_1)) \land \ldots \land (A_k = k(A_k))$. We often abbreviate $(B = 1)$ with B and $(B = 0)$ with $\neg B$, for a Boolean feature $B \in \mathbb{F}$. The set of valid configurations $\mathbb{K}$ can be also represented as a formula: $\vee_{k\in\mathbb{K}} k$.

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:

$$\theta ::= \mathrm{true} \,|\, e_{\mathbb{F}_\mathbb{Z}} \bowtie e_{\mathbb{F}_\mathbb{Z}} \,|\, \neg\theta \,|\, \theta_1 \land \theta_2 \,|\, \theta_1 \lor \theta_2, \qquad e_{\mathbb{F}_\mathbb{Z}} ::= n \mid A \mid e_{\mathbb{F}_\mathbb{Z}} \oplus e_{\mathbb{F}_\mathbb{Z}}$$

where $A \in \mathbb{F}$, $n \in \mathbb{Z}$, $\oplus \in \{+,-,*\}$, and $\bowtie \in \{=,<\}$. We will use $\theta \in \mathit{FeatExp}(\mathbb{F})$ to write presence conditions. 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 SIMPLE program family we have $\mathbb{F} = \{\mathtt{B}, \mathtt{SIZE}\}$, where $\mathrm{dom}(\mathtt{SIZE}) = [1,4]$, and $\mathbb{K} = \{\mathtt{B} \land (\mathtt{SIZE}=1), \mathtt{B} \land (\mathtt{SIZE}=2), \mathtt{B} \land (\mathtt{SIZE}=3), \mathtt{B} \land (\mathtt{SIZE}=4), \neg\mathtt{B} \land (\mathtt{SIZE}=1), \neg\mathtt{B} \land (\mathtt{SIZE}=2), \neg\mathtt{B} \land (\mathtt{SIZE}=3), \neg\mathtt{B} \land (\mathtt{SIZE}=4)\}$. For the feature expression $(\mathtt{SIZE}\leq 3)$, we have $[\![(\mathtt{SIZE}\leq 3)]\!] = \{\mathtt{B} \land (\mathtt{SIZE}=1), \mathtt{B} \land (\mathtt{SIZE}=2), \mathtt{B} \land (\mathtt{SIZE}=3), \neg\mathtt{B} \land (\mathtt{SIZE}=1), \neg\mathtt{B} \land (\mathtt{SIZE}=2), \neg\mathtt{B} \land (\mathtt{SIZE}=3)\}$. Hence, $B \land (\mathtt{SIZE}=4) \not\models (\mathtt{SIZE}\leq 3)$.

```
int x := 10, y := 0;
while (x !=0) {
    x := x-1;
    y := y+1;
    skip; }
```

(a) $\mathtt{P}_{B\wedge(\mathtt{SIZE}=1)}$(SIMPLE)

```
int x := 10, y := 0;
while (x !=0) {
    x := x-1;
    y := y-1;
    skip; }
```

(b) $\mathtt{P}_{B\wedge(\mathtt{SIZE}=4)}$(SIMPLE)

```
int x := 10, y := 0;
while (x !=0) {
    x := x-1;
    y := y+1;
    y := 0; }
```

(c) $\mathtt{P}_{\neg B\wedge(\mathtt{SIZE}=1)}$(SIMPLE)

```
int x := 10, y := 0;
while (x !=0) {
    x := x-1;
    y := y-1;
    y := 0; }
```

(d) $\mathtt{P}_{\neg B\wedge(\mathtt{SIZE}=4)}$(SIMPLE)

Fig. 3: Different variants of the program family SIMPLE from Section 2.

We consider a simple sequential non-deterministic programming language, which will be used to exemplify our work. The variables 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. The new statement "`#if` (θ) s" 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 s will be included in the variant corresponding to k. The syntax is:

$$s ::= \mathtt{skip} \mid \mathtt{x}\mathtt{:=}e \mid s;s \mid \mathtt{if}\ (e)\ \mathtt{then}\ s\ \mathtt{else}\ s \mid \mathtt{while}\ (e)\ \mathtt{do}\ s \mid \mathtt{\#if}\ (\theta)\ s\ \mathtt{\#endif},$$
$$e ::= n \mid [n, n'] \mid \mathtt{x} \mid e \oplus e$$

where n ranges over integers, $[n, n']$ over integer intervals, x over program variables, and $\oplus$ over binary arithmetic operators. Integer intervals $[n, n']$ denote a random choice of an integer in the interval. The set of all statements s is denoted by *Stm*; the set of all expressions e is denoted by *Exp*.

A program family is evaluated in two stages. First, the C *preprocessor* CPP takes a program family s and a configuration $k \in \mathbb{K}$ as inputs, and produces a variant (without `#if`-s) corresponding to k as the output. Second, the obtained variant is evaluated using the standard single-program semantics. 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 sub-statements of compound statements. Hence, $\mathtt{P}_k(\mathtt{skip}) = \mathtt{skip}$ and $\mathtt{P}_k(s;s') = \mathtt{P}_k(s);\mathtt{P}_k(s')$. The interesting case is "`#if` (θ) s", where statement s is included in the variant if $k \models \theta$, otherwise, s is removed [5]:

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

For example, variants $\mathtt{P}_{B\wedge(\mathtt{SIZE}=1)}$(SIMPLE), $\mathtt{P}_{B\wedge(\mathtt{SIZE}=4)}$(SIMPLE), $\mathtt{P}_{\neg B\wedge(\mathtt{SIZE}=1)}$(SIMPLE), as well as $\mathtt{P}_{\neg B\wedge(\mathtt{SIZE}=4)}$(SIMPLE) shown in Fig. 3a, Fig. 3b, Fig. 3c, and Fig. 3d, respectively, are derived from SIMPLE defined in Section 2.

4 Lifted Analysis based on Tuples

Lifted analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. They directly analyze

[5] Since $k \in \mathbb{K}$ is a valuation function, either $k \models \theta$ holds or $k \not\models \theta$ holds for any θ.

program families, without preprocessing them. Lifted analysis as defined by Midtgaard et. al. [19] rely on a lifted domain that is $|\mathbb{K}|$-fold product of an existing single-program analysis domain $\mathbb{A}$. We assume that the single-program domain $\mathbb{A}$ is equipped with sound operators for concretization $\gamma_{\mathbb{A}}$, ordering $\sqsubseteq_{\mathbb{A}}$, join $\sqcup_{\mathbb{A}}$, meet $\sqcap_{\mathbb{A}}$, bottom $\bot_{\mathbb{A}}$, top $\top_{\mathbb{A}}$, widening $\nabla_{\mathbb{A}}$, and narrowing $\triangle_{\mathbb{A}}$, as well as sound transfer functions for tests $\mathrm{FILTER}_{\mathbb{A}}$ and forward assignments $\mathrm{ASSIGN}_{\mathbb{A}}$. More specifically, $\mathrm{FILTER}_{\mathbb{A}}(a : \mathbb{A}, e : \mathit{Exp})$ returns an abstract element from $\mathbb{A}$ obtained by restricting a to satisfy the test e, whereas $\mathrm{ASSIGN}_{\mathbb{A}}(a : \mathbb{A}, \texttt{x:=}e : \mathit{Stm})$ returns an updated version of a by abstractly evaluating $\texttt{x:=}e$ in it.

Lifted Domain. The *lifted analysis domain* is defined as $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$, where $\mathbb{A}^{\mathbb{K}}$ is shorthand for the $|\mathbb{K}|$-fold product $\prod_{k \in \mathbb{K}} \mathbb{A}$, that is, there is one separate copy of $\mathbb{A}$ for each configuration of $\mathbb{K}$. For example, consider the tuple in Fig. 1.

Lifted Abstract Operations. Given a tuple (lifted domain element) $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, the projection π_k selects the k^{th} component of $\overline{a}$. All abstract lifted operations are defined by lifting the abstract operations of the domain $\mathbb{A}$ configuration-wise.

$$\begin{array}{ll}
\overline{\gamma}(\overline{a}) \;=\; \prod_{k \in \mathbb{K}}(\gamma_{\mathbb{A}}(\pi_k(\overline{a}))), & \overline{a_1} \dot{\sqsubseteq} \overline{a_2} \equiv \pi_k(\overline{a_1}) \sqsubseteq_{\mathbb{A}} \pi_k(\overline{a_2}), \text{for } \forall k \in \mathbb{K} \\
\overline{a_1} \;\dot{\sqcup}\; \overline{a_2} \;=\; \prod_{k \in \mathbb{K}}(\pi_k(\overline{a_1}) \sqcup_{\mathbb{A}} \pi_k(\overline{a_2})), & \overline{a_1} \;\dot{\sqcap}\; \overline{a_2} \;=\; \prod_{k \in \mathbb{K}}(\pi_k(\overline{a_1}) \sqcap_{\mathbb{A}} \pi_k(\overline{a_2})) \\
\dot{\top} \;=\; \prod_{k \in \mathbb{K}} \top_{\mathbb{A}} \;=\; (\top_{\mathbb{A}}, \ldots, \top_{\mathbb{A}}), & \dot{\bot} \;=\; \prod_{k \in \mathbb{K}} \bot_{\mathbb{A}} \;=\; (\bot_{\mathbb{A}}, \ldots, \bot_{\mathbb{A}}) \\
\overline{a_1} \;\dot{\nabla}\; \overline{a_2} \;=\; \prod_{k \in \mathbb{K}}(\pi_k(\overline{a_1}) \nabla_{\mathbb{A}} \pi_k(\overline{a_2})), & \overline{a_1} \;\dot{\triangle}\; \overline{a_2} \;=\; \prod_{k \in \mathbb{K}}(\pi_k(\overline{a_1}) \triangle_{\mathbb{A}} \pi_k(\overline{a_2}))
\end{array}$$

Lifted Transfer Functions. We now define lifted transfer functions for tests, forward assignments ($\overline{\mathrm{ASSIGN}}$), and `#if`-s ($\overline{\mathrm{IFDEF}}$). There are two types of tests: *expression-based tests*, denoted $\overline{\mathrm{FILTER}}$, that occur in `while`-s and `if`-s, and *feature-based tests*, denoted $\overline{\text{FEAT-FILTER}}$, that occur in `#if`-s. Each lifted transfer function takes as input a tuple from $\mathbb{A}^{\mathbb{K}}$ representing the invariant before evaluating the statement (resp., expression) to handle, and returns a tuple representing the invariant after evaluating the given statement (resp., expression).

$$\begin{array}{l}
\overline{\mathrm{FILTER}}(\overline{a} : \mathbb{A}^{\mathbb{K}},\, e : \mathit{Exp}) = \prod_{k \in \mathbb{K}}(\mathrm{FILTER}_{\mathbb{A}}(\pi_k(\overline{a}), e)) \\
\overline{\text{FEAT-FILTER}}(\overline{a} : \mathbb{A}^{\mathbb{K}},\, \theta : \mathit{FeatExp}(\mathbb{F})) = \prod_{k \in \mathbb{K}} \begin{cases} \pi_k(\overline{a}), & \text{if } k \models \theta \\ \bot_{\mathbb{A}}, & \text{if } k \not\models \theta \end{cases} \\
\overline{\mathrm{ASSIGN}}(\overline{a} : \mathbb{A}^{\mathbb{K}},\, \texttt{x:=}e : \mathit{Stm}) = \prod_{k \in \mathbb{K}}(\mathrm{ASSIGN}_{\mathbb{A}}(\pi_k(\overline{a}), \texttt{x:=}e)) \\
\overline{\mathrm{IFDEF}}(\overline{a} : \mathbb{A}^{\mathbb{K}}, \texttt{\#if } (\theta)\; s : \mathit{Stm}) = \overline{[\![s]\!]}(\overline{\text{FEAT-FILTER}}(\overline{a}, \theta)) \dot{\sqcup} \overline{\text{FEAT-FILTER}}(\overline{a}, \neg\theta)
\end{array}$$

where $\overline{[\![s]\!]}(\overline{a})$ is the lifted transfer function for statement s. $\overline{\mathrm{FILTER}}$ and $\overline{\mathrm{ASSIGN}}$ are defined by applying $\mathrm{FILTER}_{\mathbb{A}}$ and $\mathrm{ASSIGN}_{\mathbb{A}}$ independently on each component of the input tuple $\overline{a}$. $\overline{\text{FEAT-FILTER}}$ keeps those components k of the input tuple $\overline{a}$ that satisfy θ, otherwise it replaces the other components with $\bot_{\mathbb{A}}$. $\overline{\mathrm{IFDEF}}$ captures the effect of analyzing the statement s in the components k of $\overline{a}$ that satisfy θ, otherwise it is an identity for the other components.

Lifted Analysis. Lifted abstract operators and transfer functions of $\mathbb{A}^{\mathbb{K}}$ are combined together to analyze program families. Initially, we build a tuple $\overline{a}_{in}$ where all components are set to $\top_{\mathbb{A}}$ for the first program location, and tuples where all components are set to $\bot_{\mathbb{A}}$ for all other locations. The analysis properties are propagated forward from the first program location towards the final location taking assignments, `#if`-s, and tests into account with join and widening around `while`-s. We apply delayed widening [9], which means we start extrapolating by widening only after some fixed number of iterations we analyze the loop. The *soundness* of the lifted analysis based on $\mathbb{A}^{\mathbb{K}}$ follows immediately from the soundness of all abstract operators and transfer functions of $\mathbb{A}$ (proved in [19]).

Numerical Lifted Analysis The single-program analysis domain $\mathbb{A}$ can be instantiated by some of the well-known numerical property domains [21].

The *Interval domain* [8], denoted as $\langle I, \sqsubseteq_I \rangle$, is a *non-relational* numerical property domain that identifies the range of possible values for every variable as an interval. The property elements are: $\{[l,h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\}$.

The *Octagon domain* [20], denoted as $\langle O, \sqsubseteq_O \rangle$, is a *weakly-relational* numerical property domain, where property elements are conjunctions of linear constraints of the form $\pm \mathtt{x}_j \pm \mathtt{x}_i \geq \beta$ between variables $\mathtt{x}_i$ and $\mathtt{x}_j$, and $\beta \in \mathbb{Z}$.

The *Polyhedra domain* [12], denoted as $\langle P, \sqsubseteq_P \rangle$, is a *fully relational* numerical property domain. It expresses conjunctions of linear constraints of the form $\alpha_1 \mathtt{x}_1 + \ldots + \alpha_n \mathtt{x}_n + \beta \geq 0$, where $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are variables and $\alpha_i, \beta \in \mathbb{Z}$.

5 Lifted Analysis based on Decision Trees

In this section, we introduce a new lifted domain that relies on *decision trees*. The elements of the lifted domain are disjunctions of the leaf nodes that belong to an existing single-program analysis domain $\mathbb{A}$. The leaf nodes are separated by linear constraints over numerical features, organized in the decision nodes. Hence, we encapsulate the set of valid configurations $\mathbb{K}$ into the decision nodes of a decision tree where each top-down path represents one or several configurations from $\mathbb{K}$ that satisfy the constraints encountered along the given path. We store in each leaf node the property generated from the variants representing the corresponding configurations. We assume $\mathbb{F} = \{A_1, \ldots, A_k\}$ be a finite and totally ordered set of numerical features, such that the ordering is $A_1 > A_2 > \ldots > A_k$.

Abstract domain for decision nodes. We define the family of abstract domains for linear constraints $\mathbb{C}_{\mathbb{D}}$, which are parameterized by any of the numerical property domains $\mathbb{D}$ (intervals I, octagons O, polyhedra P). We use $C_I = \{\pm A_i \geq \beta \mid A_i \in \mathbb{F}, \beta \in \mathbb{Z}\}$ to denote the set of *interval constraints*, $C_O = \{\pm A_i \pm A_j \geq \beta \mid A_i, A_j \in \mathbb{F}, \beta \in \mathbb{Z}\}$ to denote the set of *octagonal constraints*, and $C_P = \{\alpha_1 A_1 + \ldots + \alpha_k A_k + \beta \geq 0 \mid A_1, \ldots A_k \in \mathbb{F}, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}, \gcd(|\alpha_1|, \ldots, |\alpha_k|, |\beta|) = 1\}$ to denote the set of *polyhedral constraints*. We have $C_I \subseteq C_O \subseteq C_P$.

The set $C_{\mathbb{D}}$ 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

$\langle \mathcal{P}(C_{\mathbb{D}}), \sqsubseteq_{\mathbb{D}} \rangle \underset{\alpha_{C_{\mathbb{D}}}}{\overset{\gamma_{C_{\mathbb{D}}}}{\leftrightarrows}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$, where $\mathcal{P}(C_{\mathbb{D}})$ is the power set of $C_{\mathbb{D}}$. The abstraction function $\alpha_{C_{\mathbb{D}}} : \mathcal{P}(C_{\mathbb{D}}) \to \mathbb{D}$ maps a set of interval (resp., octagon, polyhedral) constraints to an interval (resp., an octagon, polyhedral) that represents a conjunction of constraints; the concretization function $\gamma_{C_{\mathbb{D}}} : \mathbb{D} \to \mathcal{P}(C_{\mathbb{D}})$ maps an interval (resp., an octagon, a polyhedron) that represents a conjunction of constraints to a set of interval (resp., octagonal, polyhedral) constraints. We have $\gamma_{C_{\mathbb{D}}}(\top_{\mathbb{D}}) = \emptyset$ and $\gamma_{C_{\mathbb{D}}}(\bot_{\mathbb{D}}) = \{\bot_{C_{\mathbb{D}}}\}$, where $\bot_{C_{\mathbb{D}}}$ is an unsatisfiable constraint.

The negation of linear constraints is formed as: $\neg(\alpha_1 A_1 + \ldots \alpha_k A_k + \beta \geq 0) = -\alpha_1 A_1 - \ldots - \alpha_k F_k - \beta - 1 \geq 0$. For example, the negation of $A - 3 \geq 0$ is the constraint $-A + 2 \geq 0$ (i.e. $A \leq 2$). To ensure canonical representation of decision trees, a linear constraint c and its negation $\neg c$ cannot both appear as nodes in a decision tree. For example, we only keep the largest constraint with respect to $<_{C_{\mathbb{D}}}$ between c and $\neg c$. For this reason, we define the equivalence relation $\equiv_{C_{\mathbb{D}}}$ as $c \equiv_{C_{\mathbb{D}}} \neg c$. We define $\langle \mathbb{C}_{\mathbb{D}}, <_{C_{\mathbb{D}}} \rangle$ to denote $\langle C_{\mathbb{D}}/_{\equiv}, <_{C_{\mathbb{D}}} \rangle$.

The domain of decision nodes is $\mathbb{C}_{\mathbb{D}}$. 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:

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

Abstract domain for constraint-based decision trees. A *constraint-based decision tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ over the sets $\mathbb{C}_{\mathbb{D}}$ of linear constraints defined on $\mathbb{F}$ and the leaf abstract domain $\mathbb{A}$, 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 constraint-based decision trees t_1 and t_2 have decision nodes labelled with Interval linear constraints over the numeric feature `SIZE` with domain $\{1, 2, 3, 4\}$, whereas leaf nodes are Interval properties:

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

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, $\gamma_{\mathbb{T}}(t) = \overline{\gamma}_{\mathbb{T}}[\mathbb{K}](t)$. The function $\overline{\gamma}_{\mathbb{T}}$ accumulates into a set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (initially equal to $\mathbb{K} = \vee_{k \in \mathbb{K}} k$) constraints along the paths up to a leaf node:

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

Note that $k \models C$ is equivalent with $\alpha_{\mathbb{C}_{\mathbb{D}}}(\{k\}) \sqsubseteq_{\mathbb{D}} \alpha_{\mathbb{C}_{\mathbb{D}}}(C)$, where $\alpha_{\mathbb{C}_{\mathbb{D}}}(C)$ represents a conjunction of linear constraints from the set C. Therefore, we can check $k \models C$ using the abstract operation $\sqsubseteq_{\mathbb{D}}$ of the numerical domain $\mathbb{D}$.

The other binary operations of $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ are based on Algorithm 1 for *tree unification*, which finds a common refinement (labelling) of two trees t_1 and t_2 by calling function $\texttt{UNIFICATION}(t_1, t_2, \mathbb{K})$. It possibly adds new constraints as decision nodes (Lines 5–7, Lines 12–14), or removes constraints that are redundant (Lines 3,4,10,11,16,17). The function `UNIFICATION` accumulates into the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (initialized to $\mathbb{K}$, which represents implicit constraints satisfied by both t_1 and t_2), constraints encountered along the paths of the decision tree. This set C is used by the function isRedundant(c, C), which checks whether the linear constraint $c \in \mathbb{C}_{\mathbb{D}}$ is redundant with respect to C by testing $\alpha_{\mathbb{C}_{\mathbb{D}}}(C) \sqsubseteq_{\mathbb{D}} \alpha_{\mathbb{C}_{\mathbb{D}}}(\{c\})$.

```
Algorithm 1: UNIFICATION($t_1, t_2, C$)
1  if isLeaf($t_1$) $\wedge$ isLeaf($t_2$) then return ($t_1, t_2$);
2  if isLeaf($t_1$) $\vee$ (isNode($t_1$) $\wedge$ isNode($t_2$) $\wedge$ $t_2.c <_{\mathbb{C}_{\mathbb{D}} \cup \mathbb{F}_{\mathbb{B}}} t_1.c$) then
3      if isRedundant($t_2.c, C$) then return UNIFICATION($t_1, t_2.l, C$);
4      if isRedundant($\neg t_2.c, C$) then return UNIFICATION($t_1, t_2.r, C$);
5      $(l_1, l_2)$ = UNIFICATION($t_1, t_2.l, C \cup \{t_2.c\}$);
6      $(r_1, r_2)$ = UNIFICATION($t_1, t_2.r, C \cup \{\neg t_2.c\}$);
7      return ($\llbracket t_2.c : l_1, r_1 \rrbracket, \llbracket t_2.c : l_2, r_2 \rrbracket$);
8  if isLeaf($t_2$) $\vee$ (isNode($t_1$) $\wedge$ isNode($t_2$) $\wedge$ $t_1.c <_{\mathbb{C}_{\mathbb{D}} \cup \mathbb{F}_{\mathbb{B}}} t_2.c$) then
9      if isRedundant($t_1.c, C$) then return UNIFICATION($t_1.l, t_2, C$);
10     if isRedundant($\neg t_1.c, C$) then return UNIFICATION($t_1.r, t_2, C$);
11     $(l_1, l_2)$ = UNIFICATION($t_1.l, t_2, C \cup \{t_1.c\}$);
12     $(r_1, r_2)$ = UNIFICATION($t_1.r, t_2, C \cup \{\neg t_1.c\}$);
13     return ($\llbracket t_1.c : l_1, r_1 \rrbracket, \llbracket t_1.c : l_2, r_2 \rrbracket$);
14 else
15     if isRedundant($t_1.c, C$) then return UNIFICATION($t_1.l, t_2.l, C$);
16     if isRedundant($\neg t_1.c, C$) then return UNIFICATION($t_1.r, t_2.r, C$);
17     $(l_1, l_2)$ = UNIFICATION($t_1.l, t_2.l, C \cup \{t_1.c\}$);
18     $(r_1, r_2)$ = UNIFICATION($t_1.r, t_2.r, C \cup \{\neg t_1.c\}$);
19     return ($\llbracket t_1.c : l_1, r_1 \rrbracket, \llbracket t_1.c : l_2, r_2 \rrbracket$);
```

Example 2. Consider constraint-based decision trees t_1 and t_2 from Example 1. After tree unification $\texttt{UNIFICATION}(t_1, t_2, \mathbb{K})$, the resulting decision trees are:

$$t_1 = \llbracket \texttt{SIZE} \geq 4 : \ll [y \geq 2] \gg, \llbracket \texttt{SIZE} \geq 2 : \ll [y = 0] \gg, \ll [y = 0] \gg \rrbracket \rrbracket,$$
$$t_2 = \llbracket \texttt{SIZE} \geq 4 : \ll [y \geq 0] \gg, \llbracket \texttt{SIZE} \geq 2 : \ll [y \geq 0] \gg, \ll [y \leq 0] \gg \rrbracket \rrbracket$$

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

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

$$\ll a_1 \gg \sqsubseteq_{\mathbb{T}} \ll a_2 \gg = a_1 \sqsubseteq_{\mathbb{A}} a_2, \quad \llbracket c : tl_1, tr_1 \rrbracket \sqsubseteq_{\mathbb{T}} \llbracket c : tl_2, tr_2 \rrbracket = (tl_1 \sqsubseteq_{\mathbb{T}} tl_2) \wedge (tr_1 \sqsubseteq_{\mathbb{T}} tr_2)$$

while their join $t_1 \sqcup_{\mathbb{T}} t_2$ is defined as:

$$\ll\! a_1 \!\gg \sqcup_{\mathbb{T}} \ll\! a_2 \!\gg = \ll\! a_1 \sqcup_{\mathbb{A}} a_2 \!\gg, \quad [\![c\!:\!tl_1, tr_1]\!] \sqcup_{\mathbb{T}} [\![c\!:\!tl_2, tr_2]\!] = [\![c : tl_1 \sqcup_{\mathbb{T}} tl_2, tr_1 \sqcup_{\mathbb{T}} tr_2]\!]$$

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: $\bot_{\mathbb{T}} = \ll\! \bot_{\mathbb{A}} \!\gg$.

Example 3. Consider the unified trees t_1 and t_2 from Example 2. We have that $t_1 \sqsubseteq_{\mathbb{T}} t_2$ holds, and $t_1 \sqcup_{\mathbb{T}} t_2 = [\![\mathtt{SIZE} \geq 4 : \ll\![y \geq 0]\!\gg, [\![\mathtt{SIZE} \geq 2 : \ll\![y \geq 0]\!\gg, \ll\![y \leq 0]\!\gg]\!]]\!]$.

Transfer functions. The transfer functions for forward assignments ($\mathrm{ASSIGN}_{\mathbb{T}}$) and expression-based tests ($\mathrm{FILTER}_{\mathbb{T}}$) modify only leaf nodes of a constraint-based decision tree. In contrast, transfer functions for variability-specific constructs, such as feature-based tests ($\mathrm{FEAT\text{-}FILTER}_{\mathbb{T}}$) and `#if` statements ($\mathrm{IFDEF}_{\mathbb{T}}$) 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 feature variables is located in decision nodes.

Algorithm 2: $\mathtt{ASSIGN}_{\mathbb{T}}(t, \mathtt{x}\!:=\!e)$

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

The transfer function $\mathrm{ASSIGN}_{\mathbb{T}}$ for handling an assignment $\mathtt{x}\!:=\!e$ in the input tree t is described by Algorithm 2. Note that `x` is a program variable, and $e \in \mathit{Exp}$ may contain only program variables. We apply $\mathrm{ASSIGN}_{\mathbb{A}}$ to each leaf node a of t, which substitutes the expression e for the variable `x` within a. Similarly, the transfer function $\mathrm{FILTER}_{\mathbb{T}}$ for handling expression-based tests $e \in \mathit{Exp}$ is implemented by applying $\mathrm{FILTER}_{\mathbb{A}}$ leaf-wise (see Algorithm 4 in App. A).

The transfer function $\mathrm{FEAT\text{-}FILTER}_{\mathbb{T}}$ for feature-based tests θ is described by Algorithm 3. 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 $\mathrm{FILTER}_{\mathbb{D}}$ to approximate θ, thus producing a set of constraints C, which are then added to the tree t, possibly discarding all paths of t that do not satisfy θ. This is done by calling the function $\mathtt{AUGMENT}(t, \mathbb{K}, C)$, which adds linear constraints from C to t in ascending order with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ (see Algorithm 5 in App. A). 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 $\mathrm{FILTER}_{\mathbb{D}}$ 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, the transfer function $\mathrm{IFDEF}_{\mathbb{T}}$ is defined as:

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

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

```
switch $\theta$ do
    case $(e_{\mathbb{F}_\mathbb{Z}} \bowtie e_{\mathbb{F}_\mathbb{Z}}) \mid\mid (\neg(e_{\mathbb{F}_\mathbb{Z}} \bowtie e_{\mathbb{F}_\mathbb{Z}}))$ do
        $C = \texttt{FILTER}_{\mathbb{D}}(\top_{\mathbb{D}}, \theta)$; return $\texttt{AUGMENT}(t, \mathbb{K}, C)$
    case $\theta_1 \wedge \theta_2$ do
        return $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_1) \sqcap_{\mathbb{T}} \texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_2)$
    case $\theta_1 \vee \theta_2$ do
        return $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_1) \sqcup_{\mathbb{T}} \texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta_2)$
```

```
① int x := 0;
② #if (SIZE ≤ 4) x := x+1; #else x := x-1; #endif
③ #if (SIZE==3 || SIZE==4) x := x-2; #endif ④
```

Fig. 4: Code base for program family P.

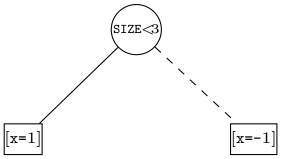

Fig. 5: Decision tree at loc. ④ of P.

where $[\![s]\!]_{\mathbb{T}}(t)$ denotes the transfer function in $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ for statement s.

After applying transfer functions, the obtained decision trees may contain some redundancy that can be exploited to further compress them. We call the function $\texttt{COMPRESS}_{\mathbb{T}}(t, \mathbb{K})$ (see Algorithm 6 in App. A) to compress the tree t. We use several optimizations. For example, if constraints on a path to some leaf are unsatisfiable, we eliminate that leaf node; if a decision node contains two same subtrees, then we keep only one subtree and we also eliminate the decision node.

Lifted analysis. The abstract operations and transfer functions of $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ can now be used to define the lifted analysis of program families. The tree t_{in} at the initial location has only one leaf node $\top_{\mathbb{A}}$ and decision nodes that define the set $\mathbb{K}$. Note that if $\mathbb{K} \equiv$ true, then $t_{in} = \top_{\mathbb{T}}$. In this way, we collect the possible invariants in the form of decision trees at all program locations.

We establish correctness of the lifted analysis based on $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ by showing that it produces identical results with tuple-based domain $\mathbb{A}^{\mathbb{K}}$. Let $[\![s]\!]_{\mathbb{T}}$ and $\overline{[\![s]\!]}$ denote transfer functions of statement s in $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ and $\mathbb{A}^{\mathbb{K}}$, respectively.

Theorem 1 (App. B). $\gamma_{\mathbb{T}}\big([\![s]\!]_{\mathbb{T}}(t_{in})\big) = \overline{\gamma}\big(\overline{[\![s]\!]}(\overline{a}_{in})\big)$.

Example 4. Let us consider the code base of a program family P given in Fig. 8. It contains only one numerical feature $\texttt{SIZE}$ with domain $\mathbb{N}$. The decision tree inferred at the final location ④ is depicted in Fig. 5. It uses the Interval domain for both decision and leaf nodes. Note that the constraint $(\texttt{SIZE} < 3)$ does not explicitly appear in the code base, but we obtain it in the decision tree representation. This shows that partitioning of the configuration space $\mathbb{K}$ induced by decision trees is semantics-based rather than syntactic-based.

6 Evaluation

Implementation We have developed a prototype lifted static analyzer, called SPLNum^{2}Analyzer, that uses lifted abstract domains of tuples $\mathbb{A}^{\mathbb{K}}$ and decision trees $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$. The abstract domains $\mathbb{A}$ for encoding properties of tuple components and leaf nodes as well as the abstract domain $\mathbb{D}$ for encoding linear constraints over numerical features are based on intervals, octagons, and polyhedra domains. Their abstract operations and transfer functions are provided by the APRON library [17]. Our proof-of-concept implementation is written in OCaml and consists of around 6K lines of code. The current front-end of the tool accepts programs written in a (subset of) C with `#if` directives, but without `struct` and `union` types. It currently provides only a limited support for arrays, pointers, and recursion. The only basic data type is mathematical integers. SPLNum^{2}Analyzer automatically infers numerical invariants in all program locations corresponding to all variants in the given family.

Experimental setup and Benchmarks All experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU, Lubuntu VM, with 8 GB memory. All times are reported as average over five independent executions. The implementation, benchmarks, and all results obtained from our experiments are available from: `https://github.com/aleksdimovski/SPLNUM2Analyzer`. In our experiments, we use three instances of our lifted analysis via decision trees: $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ that use intervals, octagons, and polyhedra domains for properties in leaf nodes and in decision nodes, respectively. We also use three instances of our lifted analysis based on tuples: $\overline{\mathcal{A}}_{\Pi}(I)$, $\overline{\mathcal{A}}_{\Pi}(O)$, and $\overline{\mathcal{A}}_{\Pi}(P)$.

SPLNum^{2}Analyzer was evaluated on a dozen of C numerical programs collected from several different folders (categories) of the 8th International Competition on Software Verification (SV-COMP 2019, `https://sv-comp.sosy-lab.org/2019/`) as well as from the real-world BusyBox project (`https://busybox.net`). The folders from SV-COMP we use are: `loops`, `loop-invgen` (`invgen` for short), `loop-lit` (`lit` for short), `termination-crafted` (`crafted` for short). In case of SV-COMP, we have first selected some numerical programs with integers, and then we have manually added variability (features and `#if` directives) in each of them. In case of BusyBox, we have first selected some programs with numerical features, and then we have simplified those programs so that our tool can handle them. For example, any reference to a pointer or a library function is replaced with $[-\infty, +\infty]$. Table 1 presents characteristics of the selected benchmarks.

Performance Results Table 1 shows the results of analyzing our benchmark files by using different versions of our lifted static analyses based on decision trees and on tuples. For each version of decision tree-based lifted analysis, there are two columns. In the first column, Time, we report the running time in seconds to analyze the given benchmark using the corresponding version of lifted analysis based on decision trees. In the second column, Impr., we report the speed up factor for each version of lifted analysis based on decision trees relative to the corresponding baseline lifted analysis based on tuples ($\overline{\mathcal{A}}_{\mathbb{T}}(I)$ vs. $\overline{\mathcal{A}}_{\Pi}(I)$, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$

Table 1: Performance results for lifted static analyses based on decision trees vs. tuples (which are used as baseline). All times are in seconds.

Bench.	folder	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{K}\rvert$	LOC	$\overline{\mathcal{A}}_{\mathbb{T}}(I)$		$\overline{\mathcal{A}}_{\mathbb{T}}(O)$		$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	
					TIME	IMPR.	TIME	IMPR.	TIME	IMPR.
`half_2.c`	`invgen`	2	36	60	0.023	2.5×	0.034	4.2×	0.037	6×
`seq.c`	`invgen`	3	125	40	0.076	11.8×	1.074	4.3×	0.249	15×
`eq1.c`	`loops`	2	36	20	0.029	4.1×	0.099	3.7×	0.084	5.2×
`sum01*.c`	`loops`	2	25	20	0.034	1.8×	0.188	1.6×	0.111	2.4×
`hhk2008.c`	`lit`	3	216	30	0.044	13.2×	0.363	4.6×	0.125	16.2×
`gcnr2008.c`	`lit`	2	25	30	0.041	2.2×	0.142	2.5×	0.165	3.4×
`Toulouse*.c`	`crafted`	3	125	75	0.084	7.7×	0.569	2.6×	0.258	10.6×
`Mysore.c`	`crafted`	3	125	35	0.039	4.6×	0.221	1.2×	0.105	6.4×
`copyfd.c`	`BusyBox`	1	16	84	0.027	4.2×	0.109	5.3×	0.113	5.2×
`real_path.c`	`BusyBox`	2	128	45	0.017	12×	0.031	20×	0.032	22×

vs. $\overline{\mathcal{A}}_{\Pi}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ vs. $\overline{\mathcal{A}}_{\Pi}(P)$). The performance results confirm that sharing is indeed effective and especially so for large values of $|\mathbb{K}|$. On our benchmarks, it translates to speed ups (i.e., ($\overline{\mathcal{A}}_{\mathbb{T}}(-)$ vs. $\overline{\mathcal{A}}_{\Pi}(-)$) that range from 1.2 to 5.2 times when $|\mathbb{K}| < 100$, and from 4.3 to 22 times when $|\mathbb{K}| > 100$.

Computational tractability The tuple-based lifted analysis $\overline{\mathcal{A}}_{\Pi}(-)$ may become very slow or even infeasible for very large configuration spaces $|\mathbb{K}|$. We have tested the limits of $\overline{\mathcal{A}}_{\Pi}(P)$ and $\overline{\mathcal{A}}_{\mathbb{T}}(-)$. We took a method, $\mathtt{test}_n^k()$, which contains n numerical features $\mathtt{A}_1, \dots, \mathtt{A}_n$, such that each numerical feature $\mathtt{A}_i$ has domain $\mathrm{dom}(\mathtt{A}_i) = [0, k-1] = \{0, \dots, k-1\}$. The body of $\mathtt{test}_n^k()$ consists of n sequentially composed `#if`-s of the form `#if` ($\mathtt{A}_i = 0$) `i := i+1 #else i := 0 #endif` For example, $\mathtt{test}_2^3()$ with two features $\mathtt{A}_1$ and $\mathtt{A}_2$, whose domain is $[0, 2]$, is:

```
①    int i := 0;
②    #if (A1 = 0) i := i+1 #else i := 0 #endif
③    #if (A2 = 0) i := i+1 #else i := 0 #endif ④
```

Subject to the chosen configuration, the variable `i` in location ④ can have a value in the range from value 2 when $\mathtt{A}_1$ and $\mathtt{A}_2$ are assigned to 0, to value 0 when $\mathtt{A}_2 \geq 1$. The analysis results in location ④ of $\mathtt{test}_2^3()$ obtained using $\overline{\mathcal{A}}_{\Pi}(P)$ and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ are shown in Fig. 6 and Fig. 7, respectively. $\overline{\mathcal{A}}_{\Pi}(P)$ uses tuples with 9 interval properties (components), while $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ uses 3 interval properties (leafs).

We have generated methods $\mathtt{test}_n^k()$ by gradually increasing variability. In general, the size of tuples used by $\overline{\mathcal{A}}_{\Pi}(P)$ is k^n, whereas the number of leaf nodes in decision trees used by $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ in the final program location is $n+1$. The performance results of analyzing $\mathtt{test}_n^k$, for different values of n and k, using $\overline{\mathcal{A}}_{\Pi}(P)$ and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ are shown in Table 2. In the columns IMPR., we report the speed-up of $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ with respect to $\overline{\mathcal{A}}_{\Pi}(P)$. Since the configurations with equivalent analysis results are nicely encoded using linear constraints in decision

$$\Big(\overbrace{[\mathtt{i}=2]}^{\mathtt{A_1=0\wedge A_2=0}},\ \overbrace{[\mathtt{i}=0]}^{\mathtt{A_1=0\wedge A_2=1}},\ \overbrace{[\mathtt{i}=0]}^{\mathtt{A_1=0\wedge A_2=2}},$$
$$\overbrace{[\mathtt{i}=1]}^{\mathtt{A_1=1\wedge A_2=0}},\ \overbrace{[\mathtt{i}=0]}^{\mathtt{A_1=1\wedge A_2=1}},\ \overbrace{[\mathtt{i}=0]}^{\mathtt{A_1=1\wedge A_2=2}},$$
$$\overbrace{[\mathtt{i}=1]}^{\mathtt{A_1=2\wedge A_2=0}},\ \overbrace{[\mathtt{i}=0]}^{\mathtt{A_1=2\wedge A_2=1}},\ \overbrace{[\mathtt{i}=0]}^{\mathtt{A_1=2\wedge A_2=2}}\Big)$$

Fig. 6: $\overline{\mathcal{A}}_{\Pi}(P)$ results at ④ of $\mathtt{test}_2^3()$.

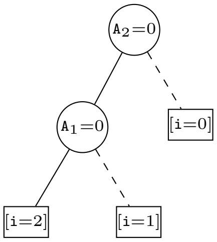

Fig. 7: $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ results at ④ of $\mathtt{test}_2^3()$.

Table 2: The performance results of analyzing $\mathtt{test}_n^k$.

n	$k = 3$			$k = 5$			$k = 7$		
	$\overline{\mathcal{A}}_{\Pi}(P)$	$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	Impr.	$\overline{\mathcal{A}}_{\Pi}(P)$	$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	Impr.	$\overline{\mathcal{A}}_{\Pi}(P)$	$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	Impr.
5	0.424	0.277	1.5×	7.078	0.278	25.4×	44.671	0.277	161.2×
6	1.869	0.591	3.1×	52.594	0.590	89.1×	infeasible	0.599	∞×
7	7.742	1.210	6.4×	infeasible	1.230	∞×	infeasible	1.221	∞×
10	584.2	10.74	54.4×	infeasible	10.826	∞×	infeasible	10.809	∞×
14	infeasible	645.10	∞×	infeasible	885.23	∞×	infeasible	898.19	∞×

nodes, the performance of $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ does not depend on k, but only depends on n. On the other hand, the performance of $\overline{\mathcal{A}}_{\Pi}(P)$ heavily depends on k.

7 Related Work and Conclusion

Decision-tree abstract domains have been used in abstract interpretation community recently [15,11,6,23]. Decision trees have been applied for the disjunctive refinement of Interval domain [15]. Segmented decision tree abstract domains has also been defined [11,6] to enable path dependent static analysis. Urban and Mine [23] use decision tree-based abstract domains to prove program termination.

Efficient implementations of the lifted dataflow analysis from *the monotone framework* have been proposed before [4,24,3]. Midtgaard et. al. [19] have proposed a formal methodology for systematic derivation of tuple-based lifted static analyses in the *abstract interpretation framework*. A more efficient lifted static analysis by abstract interpretation obtained by improving representation via BDD domains is given in [13]. However, the above lifted static analyses are applied to program families with only Boolean features. On the other hand, here we consider `#if`-enriched C families with both Boolean and numerical features, which represent the majority of industrial embedded code.

To conclude, in this work we employ decision trees and widely-known numerical abstract domains for automatic inference of invariants in all locations of C program families that contain numerical features. Using experimental evidence, we show that our lifted analysis is effective and performs well on a variety of benchmarks.

References

1. Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 372–375, 2011.
2. Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.
3. Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.
4. Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *T. Aspect-Oriented Software Development*, 10:73–108, 2013.
5. Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, 2013.*, volume 129 of *EPTCS*, pages 161–185, 2013.
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 *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
9. Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Proceedings*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.
10. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
11. 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.
12. 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 Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978.
13. Aleksandar S. Dimovski. Lifted static analysis using a binary decision diagram abstract domain. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019*, pages 102–114. ACM, 2019.
14. Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

15. Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis - 17th International Symposium, SAS 2010. Proceedings*, volume 6337 of *LNCS*, pages 287–303. Springer, 2010.
16. Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1*, pages 517–528. IEEE Computer Society, 2015.
17. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.
18. Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* PhD thesis, University of Magdeburg, Germany, May 2010.
19. 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.
20. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
21. Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017.
22. Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don S. Batory. Uniform random sampling product configurations of feature models that have numerical features. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A*, pages 39:1–39:13. ACM, 2019.
23. 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.
24. 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.

A Algorithms for Decision Tree Domains

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

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

Algorithm 5: $\texttt{AUGMENT}(t, C, J)$

```
if isEmpty($J$) then return $t$;
$j = min_{<_{\mathbb{C}_{\mathbb{D}}}}(J)$ ;
if isLeaf($t$) $\vee$ (isNode($t$) $\wedge\ j <_{\mathbb{C}_{\mathbb{D}} \cup \mathbb{F}_{\mathbb{B}}} t.c$) then
    if isREDUNDANT($j, C$) then return $\texttt{AUGMENT}(t, C, J \backslash \{j\})$;
    return ($[\![j : \texttt{AUGMENT}(t, C \cup \{j\}, J \backslash \{j\}), \texttt{AUGMENT}(t, C \cup \{\neg j\}, J \backslash \{j\})]\!]$);
else
    if (isNode($t$) $\wedge\ t.c <_{\mathbb{C}_{\mathbb{D}} \cup \mathbb{F}_{\mathbb{B}}} j$) then
        if isREDUNDANT($t.c, C$) then return $\texttt{AUGMENT}(t.l, C, J)$;
        $l = \texttt{AUGMENT}(t.l, C \cup \{t.c\}, J)$ ;
        $r = \texttt{AUGMENT}(t.r, C \cup \{\neg t.c\}, J)$ ;
        return ($[\![t.c : l, r]\!]$);
    else
        if isREDUNDANT($t.c, C$) then return $\texttt{AUGMENT}(t.l, C, J \backslash \{j\})$;
        $l = \texttt{AUGMENT}(t.l, C \cup \{t.c\}, J \backslash \{j\})$ ;
        $r = \texttt{AUGMENT}(t.r, C \cup \{\neg t.c\}, J \backslash \{j\})$ ;
        return ($[\![t.c : l, r]\!]$);
```

Algorithm 6: $\mathtt{COMPRESS}_{\mathbb{T}}(t, C)$

```
switch $t$ do
    case $\ll n \gg$ do
        return $\ll n \gg$;
    case $[\![ t.c : l, r ]\!]$ do
        $l' = \mathtt{COMPRESS}_{\mathbb{T}}(t.l, C \cup \{t.c\})$ ;
        $r' = \mathtt{COMPRESS}_{\mathbb{T}}(t.r, C \cup \{\neg t.c\})$ ;
        switch $l', r'$ do
            case $\ll n'_l \gg, \ll n'_r \gg$ when $n'_l = n'_r$ do
                return $\ll n'_l \gg$;
            case $\ll n'_l \gg, \ll n'_r \gg$ do
                if UNSAT$(C \cup \{t.c\})$ then return $\ll n'_r \gg$;
                if UNSAT$(C \cup \{\neg t.c\})$ then return $\ll n'_l \gg$;
            case $[\![ c_1 : l_1, r_1 ]\!], [\![ c_2 : l_2, r_2 ]\!]$ when $c_1 = c_2 \wedge l_1 = l_2 \wedge r_1 = r_2$ do
                return $[\![ c_1 : l_1, r_1 ]\!]$;
            case $\ll n'_l \gg, [\![ c_2 : l_2, r_2 ]\!]$ when $\ll n'_l \gg = l_2 \wedge c \leq c_2$ do
                return $[\![ c_2 : l_2, r_2 ]\!]$;
            case $[\![ c_1 : l_1, r_1 ]\!], \ll n'_r \gg$ when $\ll n'_r \gg = r_1 \wedge c_1 \leq c$ do
                return $[\![ c_1 : l_1, r_1 ]\!]$;
            case default: do
                return $[\![ t.c : l', r' ]\!]$;
```

B Proof of Theorem 1

The proof is by induction on the structure of s. Assume $\gamma_{\mathbb{T}}(t) = \overline{\gamma}(\overline{a})$ (*). We consider the two most interesting cases.

Case `x:=`e**.** $\overline{\text{ASSIGN}}(\overline{a},\ \texttt{x:=}e)$ applies $\texttt{ASSIGN}_{\mathbb{A}}(t, \texttt{x:=}e)$ to each component of $\overline{a}$. On the other hand, $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}e)$ applies $\texttt{ASSIGN}_{\mathbb{A}}(t, \texttt{x:=}e)$ to each leaf a in t. The proof follows by correctness of the assumption (*).

Case `#if` (θ) s `#endif`**.** Transfer functions for `#if` are identical in both lifted domains. We only need to show that $\overline{\text{FEAT-FILTER}}(\overline{a},\ \theta)$ and $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta)$ are identical. This can be shown by induction on θ. Assume that θ is an atomic constraint. $\overline{\text{FEAT-FILTER}}(\overline{a},\ \theta)$ keeps only those components k of $\overline{a}$ such that $k \models \theta$. On the other hand, $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta)$ first produces all linear constraints in $\mathbb{D}$ that satisfy θ, and then adds them in the tree t. Thus, it keeps only those leaf nodes that satisfy the newly generated constraints from θ. The other cases are similar.