Generalized Program Sketching by Abstract Interpretation and Logical Abduction

Aleksandar S. Dimovski[1 [0000−0002−3601−2631]]

Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia
aleksandar.dimovski@unt.edu.mk https://aleksdimovski.github.io/

Abstract. This paper presents a new approach for synthesizing missing parts from imperative programs by using abstract interpretation and logical abduction. Given a partial program with missing arbitrary expressions, our approach synthesizes concrete expressions that are strong enough to prove the assertions in the given program. Furthermore, the synthesized elements by our approach are the simplest and the weakest among all possible that guarantee the validity of assertions. In particular, we use a combination of forward and backward numerical analyses based on abstract interpretation to generate constraints that are solved by using the logical abduction technique.
We have implemented our approach in a prototype synthesis tool for C programs, and we show that the proposed approach is able to successfully synthesize arithmetic and boolean expressions for various C programs.

Keywords: Program Synthesis · Abstract interpretation · Logical Abduction.

1 Introduction

Program synthesis [2] is a task of inferring a program that satisfies a given specification. A sketch [32,33] is a partial program with missing arithmetic and boolean expressions called *holes*, which need to be discovered by the synthesizer. Previous approaches for program sketching [19,32,33] automatically synthesize only integer "constants" for the holes so that the resulting complete program satisfies given assertions for all possible inputs. However, it is more challenging to define a synthesis algorithm that infers arbitrary arithmetic and boolean expressions in program sketches. We refer to this as *generalized sketching problem*.

In this paper, we propose a new approach for solving the generalized sketching problem by using abstract interpretation [5,29,34] and logical abduction [1,8,12].

Assume that we have a program sketch with an unknown expression indicated by ??. Our synthesis algorithm computes an expression E over program variables such that, when ?? is replaced by E, all assertions within the resulting complete program become valid. Our synthesis algorithm proceeds in two phases, consisting of constraint generation and constraint solving. The constraint generation phase is based on abstract interpretation, whereas the constraint solving phase is based on logical abduction. In particular, we use forward and backward static analyses based on abstract interpretation to simultaneously compute the invariant post-condition before and sufficient precondition that ensures assertion validity after the unknown hole. The forward analysis computes an invariant P representing the facts known at the program location before the hole, whereas the backward analysis provides a sufficient precondition C that guarantees that the code after the hole satisfies all assertions. Then, we use abduction to find missing hypothesis in a logical inference task. In more detail, assume we have a premise P and a desired conclusion C for an inference, where P and C are constraints generated using forward and backward analyses, respectively. The abduction infers the simplest and most general explanation E such that $P \wedge E \models C$ and $P \wedge E \not\models \text{false}$. The first condition states that the abduction solution E together with premise P should imply conclusion C, while the second condition states that the abduction solution E should not contradict premise P. Finally, we use explanation E to synthesize a concrete expression for hole ??.

We have implemented our approach in a prototype program synthesizer. Our tool uses the numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library [25] for static analysis, as well as the EXPLAIN tool [8] for logical abduction in the combination SMT theory of linear arithmetic and propositional logic, and the MISTRAL tool [9] for SMT solving. We illustrate this approach for automatic completion of various numerical C sketches from the SKETCH project [32,33], SV-COMP (`https://sv-comp.sosy-lab.org/`), and the literature [28]. We compare performances of our approach against the most popular sketching tool SKETCH [32,33] and the FAMILYSKETCHER [19,15] that are used for synthesizing program sketches with missing integer constants.

This work makes several contributions:

(1) We explore the idea of automatically synthesizing arbitrary expressions in program sketches;
(2) We show how this generalized program sketching problem can be solved using abstract interpretation and logical abduction;
(3) We build a synthesizer using tools for static analysis by abstract interpretation and logical abduction, and present the synthesis results for the domain of numerical (linear arithmetic) programs.

2 Motivating Examples

We now present an overview of our approach using motivating examples. Consider the code `intro.c` shown in Fig. 1, where the unknown hole `??` is an arithmetic

```
void main(int x){
① int z = x+1;
② int y = z;
③ y = ??;
④ z = z+[2,3];
⑤ assert(z>y); }
```

Fig. 1. `intro.c`.

```
void main(int n){
① int abs = n;
② if (??) ⓣ abs = -n;
③ else ⓔ skip;
④ assert(abs ≥ 0); }
```

Fig. 2. `abs.c`.

```
void main() {
① int x = 10, y = 0;
② while ⓗ (??) {
③   x = x-1;
④   y = y+1; }
⑤ assert(y==10); }
```

Fig. 3. `while.c`.

expression in an assignment. The goal is to complete the unknown hole in loc. ③ so that the assertion is always valid.

Our approach starts by performing forward and backward static analyses that compute numerical invariants and sufficient conditions. They are abstract interpretation-based static analyses implemented using the Polyhedra abstract domain [7] from the APRON library [25]. The (over-approximating) forward analysis infers the invariant (`z=x+1` $\wedge$ `y=x+1`) at loc. ③ before the hole. The subsequent (under-approximating) backward analysis starts with the assertion fact (`z>y`) at loc. ⑤, and by propagating it backwards it infers the precondition (`z+2>y`) at loc. ④ after the hole. [1] We use these inferred facts to construct the following abduction query:

$$(\texttt{z=x+1} \wedge \texttt{y'=x+1}) \wedge R(\texttt{y,y',x,z}) \implies (\texttt{z+2>y})$$

which is solved by calling the EXPLAIN tool [8]. The left-hand side of the implication encodes the generated constraints up to the hole `??`, where `y'` denotes the value of variable `y` before loc. ③ and the unknown predicate R(`y,y',x,z`) encodes the constraint over all program variables in scope at loc. ③. The right-hand side of this implication encodes the postcondition ensuring that the assertion must be valid. Hence, this abduction query is looking for a constraint over program variables that guarantees the validity of this implication. Among the class of all solutions, we prefer abductive solutions containing the variable `y` over others. For this reason, we specify in the abduction query the lowest cost to variable `y`, while `y', x` and `z` have higher costs. The logically weakest and simplest [2] solution containing `y` is: R(`y,y',x,z`) $\equiv$ (`y` $\leq$ `y'+1`). This represents a weakest specification for the hole, so we can use it to synthesize the unknown expression. That is, we can fill the hole in loc. ③ with: `y = y+1` (other solutions are also possible, e.g. `y = y-1`).

Consider `abs.c` shown in Fig. 2, where the unknown hole `??` is a boolean expression in the `if`-guard. The forward analysis infers the invariant (`abs=n`) at loc. ②. The backward analysis starts with the assertion satisfaction, i.e. by propagating the assertion fact (`abs` $\geq$ 0) backwards. After the `if`-guard, it

[1] This condition guarantees that the assertion is valid for any non-deterministic choice $[2, 3]$. If we have used an over-approximating backward analysis, it would infer the necessary condition (`z+3>y`) that may lead to the assertion satisfaction for some non-deterministic choices of $[2, 3]$ (e.g., the execution where the non-deterministic choice $[2, 3]$ returns 3).

[2] A solution is simplest if it contains the fewest number of variables.

infers that the precondition of `then` branch is $(\texttt{n} \leq 0)$ and the precondition of `else` branch is $(\texttt{abs} \geq 0)$. Thus, we construct two abduction queries: (1) $(\texttt{abs=n}) \wedge R_{true}(\texttt{n,abs}) \implies (\texttt{n} \leq 0)$, and (2) $(\texttt{abs=n}) \wedge R_{false}(\texttt{n,abs}) \implies (\texttt{abs} \geq 0)$, which are solved by EXPLAIN tool [8]. The reported weakest solutions are: $R_{true}(\texttt{n,abs}) \equiv (\texttt{n} \leq 0)$ and $R_{false}(\texttt{n,abs}) \equiv (\texttt{n} \geq 0)$. Subsequently, we check whether $\neg R_{true}(\texttt{n,abs}) \implies R_{false}(\texttt{n,abs})$ using the MISTRAL SMT solver [9]. Since $\neg(\texttt{n} \leq 0) \implies (\texttt{n} \geq 0)$ is valid, we fill the hole at loc. ② with the boolean guard $(\texttt{n} \leq 0)$ as a sufficient condition for the assertion to hold.

Consider the code `while.c` given in Fig. 3, where the unknown hole `??` is a boolean expression in the `while`-guard. The forward analysis infers the invariant $(\texttt{x} \leq 10) \wedge (\texttt{x+y=10})$ at loc. ⓗ. We construct the abduction query $(\texttt{x} \leq 10) \wedge (\texttt{x+y=10}) \wedge R_{false}(\texttt{x,y}) \implies (\texttt{y=10})$. The reported solution by EXPLAIN is $R_{false}(\texttt{x,y}) \equiv (\texttt{x=0})$. We compute $R_{true}(\texttt{x,y}) \equiv \neg R_{false}(\texttt{x,y}) \equiv (\texttt{x<0}) \vee (\texttt{x>0})$. Hence, we perform two backward analysis of `while.c` in which the `while`-guard is $(\texttt{x<0} \wedge \texttt{??}_1)$ and $(\texttt{x>0} \wedge \texttt{??}_2)$, respectively. They start with the fact $(\texttt{x=0} \wedge \texttt{y=10})$ at loc. ⑤. The first backward analysis for $(\texttt{x<0} \wedge \texttt{??}_1)$ infers the bottom $(\bot)$ condition after the guard at loc. ③, whereas the second backward analysis for $(\texttt{x>0} \wedge \texttt{??}_2)$ infers $(1 \leq \texttt{x} \leq 11) \wedge (\texttt{x+y=10})$ after the guard at loc. ③. We construct two abduction queries: (1) $(\texttt{x} \leq 10 \wedge \texttt{x+y=10}) \wedge R^1_{true}(\texttt{x,y}) \implies$ false, and (2) $(\texttt{x} \leq 10 \wedge \texttt{x+y=10}) \wedge R^2_{true}(\texttt{x,y}) \implies (1 \leq \texttt{x} \leq 11) \wedge (\texttt{x+y=10})$. The obtained solutions are $R^1_{true}(\texttt{x,y}) \equiv$ false and $R^2_{true}(\texttt{x,y}) \equiv$ true. Hence, we fill the hole with $(\texttt{x>0})$.

Assume that the assertion at loc. ⑤ of `while.c` is $\texttt{assert}\,(\texttt{y} \leq 10)$. In this case, the solution of the first abduction query is $R_{false}(\texttt{x,y}) \equiv (\texttt{x} \geq 0)$. We find one interpretation $(\texttt{x=n})$, where $\texttt{n} \geq 0$, of the formula $(\texttt{x} \geq 0)$ using MISTRAL SMT solver. So, we obtain $R_{true}(\texttt{x,y}) \equiv \neg R_{false}(\texttt{x,y}) \equiv (\texttt{x<n}) \vee (\texttt{x>n})$. Then, we proceed analogously to above (basically replacing $(\texttt{x=0})$ by $(\texttt{x=n})$).

3 Language and Semantics

This section introduces the target language of our approach as well as its concrete and abstract semantics. They will be employed for designing the invariance (reachability) and sufficient condition static analyses using the abstract interpretation theory [5,29,34]. Moreover, we formally define the logical abduction problem.

3.1 Syntax

We use a simple C-like imperative language for writing general-purpose programs. The program variables *Var* are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. To encode unknown holes, we use the construct `??`. The hole constructs $\texttt{??}_i$ are placeholders that the synthesizer must replace with suitable (arithmetic and boolean) expressions, such that the resulting program will avoid any assertion failures. The syntax is given below.

$s\,(s \in \mathit{Stm}) ::= \texttt{skip} \mid \texttt{x=}ae \mid s; s \mid \texttt{if}\,(be)\,s\,\texttt{else}\,s \mid \texttt{while}\,(be)\,\texttt{do}\,s \mid \texttt{assert}(be),$
$ae\,(ae \in \mathit{AExp}) ::= \texttt{??}_i \mid ae', \quad ae' ::= n \mid [n, n'] \mid \texttt{x} \in \mathit{Var} \mid ae' \oplus ae',$
$be\,(be \in \mathit{BExp}) ::= \texttt{??}_i \mid be', \quad be' ::= ae \bowtie ae \mid \neg be' \mid be' \wedge be' \mid be' \vee be'$

$$\overrightarrow{[\![\texttt{skip}]\!]}S = S$$
$$\overrightarrow{[\![\texttt{x = } ae]\!]}S = \{\sigma[\texttt{x} \mapsto n] \mid \sigma \in S, n \in [\![ae]\!]\sigma\}$$
$$\overrightarrow{[\![s_1 \texttt{ ; } s_2]\!]}S = \overrightarrow{[\![s_2]\!]}(\overrightarrow{[\![s_1]\!]}S)$$
$$\overrightarrow{[\![\texttt{if } be \texttt{ } s_1 \texttt{ else } s_2]\!]}S = \overrightarrow{[\![s_1]\!]}\{\sigma \in S \mid \text{true} \in [\![be]\!]\sigma\} \cup \overrightarrow{[\![s_2]\!]}\{\sigma \in S \mid \text{false} \in [\![be]\!]\sigma\}$$
$$\overrightarrow{[\![\texttt{while } be \texttt{ do } s]\!]}S = \{\sigma \in \texttt{lfp } \overrightarrow{\phi} \mid \text{false} \in [\![be]\!]\sigma\}$$
$$\overrightarrow{\phi}(X) = S \cup \overrightarrow{[\![s]\!]}\{\sigma \in X \mid \text{true} \in [\![be]\!]\sigma\}$$

Fig. 4. Definitions of $\overrightarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$.

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, $\texttt{x}$ over program variables *Var*, and $\oplus \in \{+, -, *, /\}$, $\bowtie \in \{<, \leq, =, \neq\}$. Integer intervals $[n, n']$ denote a random choice of an integer in the interval. We assume that statements and holes are tagged with unique syntactic labels $\textcircled{i} \in \mathbb{L}$. Without loss of generality, we assume that a program p is a sequence of statements followed by a single assertion "$\textcircled{i}$ s; $\textcircled{f}$ $\texttt{assert}$ (be^f)".

The unknown holes $\texttt{??}_i$ occur either as tests (boolean expressions) in `if` and `while` statements or as right-hand sides (arithmetic expressions) in assignments. Let H be a set of uniquely labelled holes $\texttt{??}_i$ in program p. A *control function* ϕ is a mapping from the set of holes H to concrete (arithmetic and boolean) expressions. We say that ϕ is *complete* if $dom(\phi) = H$, i.e. ϕ is defined for each hole in the program. Otherwise, if $dom(\phi) \subset H$, i.e. ϕ is $\perp$ (undefined) for some holes, we say that ϕ is a *partial* control function. We write $p[\phi]$ to denote the program obtained by substituting each $\texttt{??}_i$ with $\phi(\texttt{??}_i)$, if $\phi(\texttt{??}_i)$ is defined.

Definition 1. *A complete control function ϕ is a* solution *to the generalized sketching problem defined by program p if $p[\phi]$ is a complete program that satisfies all its assertions under all possible inputs.*

3.2 Concrete Semantics and Analyses

We now define the concrete semantics of our language, and use it to construct *invariance* (forward) and *sufficient condition* (backward) *concrete analyses*. Such analyses are obviously *uncomputable*, since our language is Turing complete. In the next subsection, we present their sound decidable abstractions, which can statically determine dynamic properties of programs.

A memory *store*, denoted $\sigma \in \Sigma$, is a function mapping each variable to a value: $\Sigma = Var \to \mathbb{Z}$. The concrete domain is the powerset complete lattice $\langle \mathcal{P}(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma \rangle$. The semantics of arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$ and boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\text{true}, \text{false}\})$ are the sets of possible (numerical and boolean) values for expressions ae and be in a given store σ. For example, $[\![\texttt{??}_i]\!]\sigma = \mathbb{Z}$ and $[\![[n, n']]\!]\sigma = \{n, \ldots, n'\}$ for arithmetic expressions $\texttt{??}_i$ and $[n, n']$. We consider two semantics of statements (programs): an *invariance* (forward) semantics $\overrightarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ that infers a set of reachable stores (invariants) from a given set of initial stores; and a *sufficient condition* (backward)

$$
\begin{array}{l}
\overleftarrow{[\![\texttt{skip}]\!]}S = S \\
\overleftarrow{[\![\texttt{x = } ae]\!]}S = \{\sigma \mid \forall n \in [\![ae]\!]\sigma, \sigma[\texttt{x} \mapsto n] \in S\} \\
\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]}S = \overleftarrow{[\![s_1]\!]}(\overleftarrow{[\![s_2]\!]}S) \\
\overleftarrow{[\![\texttt{if } be \texttt{ } s_1 \texttt{ else } s_2]\!]}S = \big(\overleftarrow{[\![s_1]\!]}S \cup \{\sigma \mid [\![be]\!]\sigma = \{\text{false}\}\}\big) \cap \\
\qquad\qquad\qquad\qquad\qquad \big(\overleftarrow{[\![s_2]\!]}S \cup \{\sigma \mid [\![\neg be]\!]\sigma = \{\text{false}\}\}\big) \\
\overleftarrow{[\![\texttt{while } be \texttt{ do } s]\!]}S = \texttt{gfp}\ \overleftarrow{\phi} \\
\overleftarrow{\phi}(X) = \big(S \cup \{\sigma \mid [\![\neg be]\!]\sigma = \{\text{false}\}\}\big) \cap \big(\overleftarrow{[\![s]\!]}X \cup \{\sigma \mid [\![be]\!]\sigma = \{\text{false}\}\}
\end{array}
$$

Fig. 5. Definitions of $\overleftarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$.

semantics $\overleftarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ that infers a set of stores (sufficient condition) from which only stores satisfying a given postcondition are reached. The definitions of $\overrightarrow{[\![s]\!]}$ and $\overleftarrow{[\![s]\!]}$ are given in Fig. 4 and Fig. 5, respectively. The invariance semantics [5] is built forward, so each function $\overrightarrow{[\![s]\!]}$ takes as input a set of stores S before statement s and returns a set of possible stores reached after executing s from S. The sufficient condition semantics [29] is built backward, so each function $\overleftarrow{[\![s]\!]}$ takes as input a set of stores S after statement s and returns a set of possible stores before s from which only stores from S are reached after executing s. The semantics of a `while` statement is given in a standard fixed-point formulation [5,29] using the least and greatest fix-point operators `lfp` and `gfp`, where the fixed-point functionals $\overrightarrow{\phi}, \overleftarrow{\phi} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ accumulate possible stores after another `while` iteration from a set of stores X going in a forward and backward direction, respectively.

Assume that a program "ⓘ s; ⓕ `assert` (be^f)" is given. We can use the invariance semantics $\overrightarrow{[\![s]\!]}$ to collect the possible stores in all program locations reachable from a set of initial stores $\mathcal{I} \subseteq \mathcal{P}(\Sigma)$, denoted $\texttt{Inv}_{\mathcal{I}}$. We can also use the sufficient condition semantics $\overleftarrow{[\![s]\!]}$ to infer sufficient conditions in the form of a set of possible stores in all program locations that guarantee the stores after executing the program belong to some user-supplied property $\mathcal{F} \subseteq \mathcal{P}(\Sigma)$, denoted $\texttt{Cond}_{\mathcal{F}}$. We assume that at the initial label ⓘ the set of possible stores is $\mathcal{I} = \mathcal{P}(\Sigma)$, whereas at the final (assertion) label ⓕ the possible stores are $\mathcal{F} = \{\sigma \in \texttt{Inv}_{\mathcal{I}}(ⓕ) \mid [\![be^f]\!]\sigma = \{\text{true}\}\}$. That is, $\texttt{Inv}_{\mathcal{I}}(ⓘ) = \mathcal{I}$ and $\texttt{Cond}_{\mathcal{F}}(ⓕ) = \mathcal{F}$. For each statement "ⓛ s ⓛ' ", we define:

$$\texttt{Inv}_{\mathcal{I}}(ⓛ') = \overrightarrow{[\![s]\!]}\ \texttt{Inv}_{\mathcal{I}}(ⓛ), \qquad \texttt{Cond}_{\mathcal{F}}(ⓛ) = \overleftarrow{[\![s]\!]}\ \texttt{Cond}_{\mathcal{F}}(ⓛ')$$

3.3 Abstract Semantics and Analyses

We now define computable abstract analyses that are approximations of the concrete semantics and analyses. We replace the computation in the concrete domain $\mathcal{P}(\Sigma)$ with a computation in some numerical abstract domain $\mathbb{D}$ that reasons on the numerical properties of variables, such that there exists a concretization-

based abstraction $\langle \mathcal{P}(\Sigma), \subseteq \rangle \overleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$. [3] The abstract domain $\mathbb{D}$ is a set of computer-representable properties, called *abstract elements*, together with effective algorithms to implement sound abstract operators for forward and backward analyses. In particular, they have abstract operators for ordering $\sqsubseteq_{\mathbb{D}}$, least upper bound (join) $\sqcup_{\mathbb{D}}$, greatest lower bound (meet) $\sqcap_{\mathbb{D}}$, bottom $\bot_{\mathbb{D}}$, top $\top_{\mathbb{D}}$, widening $\triangledown_{\mathbb{D}}$, and narrowing $\triangle_{\mathbb{D}}$. There are forward transfer functions for assignments $\texttt{ASSIGN}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$ and tests $\texttt{FILTER}_{\mathbb{D}} : \mathit{BExp} \times \mathbb{D} \to \mathbb{D}$, which are sound over-approximations of the corresponding concrete functions. We let $\texttt{lfp}^{\#}$ denote an abstract fix-point operator, derived using widening $\triangledown_{\mathbb{D}}$ and narrowing $\triangle_{\mathbb{D}}$, that over-approximates the concrete `lfp`. There are also backward transfer functions for assignments $\texttt{B-ASSIGN}^{u}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$, tests $\texttt{B-FILTER}^{u}_{\mathbb{D}} : \mathit{BExp} \times \mathbb{D} \to \mathbb{D}$, and a lower widening $\underline{\triangledown}_{\mathbb{D}}$ [29], which are sound under-approximations of the corresponding concrete functions. We let $\texttt{gfp}^{\#}$ denote an abstract fixpoint operator, derived using lower widening $\underline{\triangledown}_{\mathbb{D}}$, that under-approximates the concrete `gfp`.

The operators of the abstract domain $\mathbb{D}$ can be used to define abstract invariance (resp., sufficient condition) analysis that is over- (resp., under-) approximation of the corresponding concrete analysis. For each statement s, we define its abstract invariance semantics $\overrightarrow{[\![s]\!]}^{\sharp}$ in Fig. 6 and its abstract sufficient condition semantics $\overleftarrow{[\![s]\!]}^{\sharp}$ in Fig. 7. For a `while` loop, $\texttt{lfp}^{\sharp}\, \overrightarrow{\phi}^{\sharp}$ and $\texttt{gfp}^{\sharp}\, \overleftarrow{\phi}^{\sharp}$ are the limits of the following increasing and decreasing chains: $y_0 = d,\ y_{n+1} = y_n \triangledown_{\mathbb{D}} \overrightarrow{\phi}^{\sharp}(y_n)$ for forward analysis; and $y_0 = \texttt{B-FILTER}^{u}_{\mathbb{D}}(\neg be, d),\ y_{n+1} = y_n \underline{\triangledown}_{\mathbb{D}} \overleftarrow{\phi}^{\sharp}(y_n)$ for backward analysis. Since $\texttt{FILTER}_{\mathbb{D}}(be, d) \sqsubseteq_{\mathbb{D}} d$ and $\texttt{B-FILTER}^{u}_{\mathbb{D}}(be, d) \sqsupseteq_{\mathbb{D}} d$, we use $\texttt{FILTER}_{\mathbb{D}}(\texttt{??}_i, d) = d$ and $\texttt{B-FILTER}^{u}_{\mathbb{D}}(\texttt{??}_i, d) = d$ to handle holes $\texttt{??}_i$ as boolean expressions. We also use $\texttt{ASSIGN}_{\mathbb{D}}(\texttt{x = ??}_i, d) = \texttt{ASSIGN}_{\mathbb{D}}(\texttt{x = } [-\infty, +\infty], d)$ and $\texttt{B-ASSIGN}^{u}_{\mathbb{D}}(\texttt{x = ??}_i, d) = \texttt{B-ASSIGN}^{u}_{\mathbb{D}}(\texttt{x = } [-\infty, +\infty], d)$ to handle holes $\texttt{??}_i$ as arithmetic expressions.

Given a program "ⓘ s; ⓕ `assert` (be^f)", we assume that at the initial label ⓘ the set of possible stores is described by abstract element $d_{\mathcal{I}}$, whereas at the final label ⓕ the user-supplied property is described by abstract element $d_{\mathcal{F}}$. That is, $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓘ) = d_{\mathcal{I}}$ and $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(ⓕ) = d_{\mathcal{F}}$. Note that $d_{\mathcal{I}} = \top_{\mathbb{D}}$ and $d_{\mathcal{F}} = \texttt{FILTER}_{\mathbb{D}}(be^f, \texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓕ))$. For each statement "ⓛ s ⓛ′", we define:

$$\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓛ') = \overrightarrow{[\![s]\!]}^{\sharp}\ \texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓛ), \qquad \texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(ⓛ) = \overleftarrow{[\![s]\!]}^{\sharp}\ \texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(ⓛ')$$

By using the soundness of the operators of abstract domain $\mathbb{D}$ [5,29], we prove the soundness of abstract semantics with respect to concrete semantics. That is, $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}$ is an over-approximation and contains some spurious stores that are not reachable in the concrete $\texttt{Inv}_{\mathcal{I}}$, whereas $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}$ is an under-approximation and does not contain some stores that are present in the concrete $\texttt{Cond}_{\mathcal{F}}$.

Proposition 1 ([5,29]). *Let $\mathcal{F} = \gamma_{\mathbb{D}}(d_{\mathcal{F}})$ and $\mathcal{I} = \gamma_{\mathbb{D}}(d_{\mathcal{I}})$. For all $l \in \mathbb{L}$, we have: $\mathit{Inv}_{\mathcal{I}}(l) \subseteq \mathit{Inv}^{\sharp}_{d_{\mathcal{I}}}(l)$, $\mathit{Cond}_{\mathcal{F}}(l) \supseteq \mathit{Cond}^{\sharp}_{d_{\mathcal{F}}}(l)$.*

[3] Concretization-based abstraction is a relaxation of the known Galois connection, which uses only a concretization function $\gamma_{\mathbb{D}}$ (e.g. Polyhedra domain).

$$
\begin{array}{l}
\overrightarrow{[\![\texttt{skip}]\!]^\sharp} d = d \\
\overrightarrow{[\![\texttt{x = } ae]\!]^\sharp} d = \texttt{ASSIGN}_{\mathbb{D}}(\texttt{x = } ae, d) \\
\overrightarrow{[\![s_1 \texttt{ ; } s_2]\!]^\sharp} d = \overrightarrow{[\![s_2]\!]^\sharp}(\overrightarrow{[\![s_1]\!]^\sharp} d) \\
\overrightarrow{[\![\texttt{if } be \texttt{ } s_1 \texttt{ else } s_2]\!]^\sharp} d = \overrightarrow{[\![s_1]\!]^\sharp}(\texttt{FILTER}_{\mathbb{D}}(be, d)) \sqcup_{\mathbb{D}} \overrightarrow{[\![s_2]\!]^\sharp}(\texttt{FILTER}_{\mathbb{D}}(\neg be, d)) \\
\overrightarrow{[\![\texttt{while } be \texttt{ do } s]\!]^\sharp} d = \texttt{FILTER}_{\mathbb{D}}(\neg be, \texttt{lfp}^\sharp \overrightarrow{\phi^\sharp}) \\
\overrightarrow{\phi^\sharp}(x) = d \sqcup_{\mathbb{D}} \overrightarrow{[\![s]\!]^\sharp}(\texttt{FILTER}_{\mathbb{D}}(be, x))
\end{array}
$$

Fig. 6. Definitions of $\overrightarrow{[\![s]\!]^\sharp} : \mathbb{D} \to \mathbb{D}$.

$$
\begin{array}{l}
\overleftarrow{[\![\texttt{skip}]\!]^\sharp} d = d \\
\overleftarrow{[\![\texttt{x = } ae]\!]^\sharp} d = \texttt{B-ASSIGN}^u_{\mathbb{D}}(\texttt{x = } ae, d) \\
\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]^\sharp} d = \overleftarrow{[\![s_1]\!]^\sharp}(\overleftarrow{[\![s_2]\!]^\sharp} d) \\
\overleftarrow{[\![\texttt{if } be \texttt{ } s_1 \texttt{ else } s_2]\!]^\sharp} d = \texttt{B-FILTER}^u_{\mathbb{D}}(be, \overleftarrow{[\![s_1]\!]^\sharp} d) \sqcap_{\mathbb{D}} \texttt{B-FILTER}^u_{\mathbb{D}}(\neg be, \overleftarrow{[\![s_2]\!]^\sharp} d) \\
\overleftarrow{[\![\texttt{while } be \texttt{ do } s]\!]^\sharp} d = \texttt{gfp}^\sharp \overleftarrow{\phi^\sharp} \\
\overleftarrow{\phi^\sharp}(x) = \texttt{B-FILTER}^u_{\mathbb{D}}(\neg be, d) \sqcap_{\mathbb{D}} \texttt{B-FILTER}^u_{\mathbb{D}}(be, \overleftarrow{[\![s]\!]^\sharp} x)
\end{array}
$$

Fig. 7. Definitions of $\overleftarrow{[\![s]\!]^\sharp} : \mathbb{D} \to \mathbb{D}$.

3.4 Abduction

The standard abduction allows the inference of a single unknown predicate $R(\mathbf{x})$ defined over a vector of variables $\mathbf{x}$, known as *abducible*, from a formula $R(\mathbf{x}) \wedge \chi \implies C$. That is, the standard abduction finds a formula ϕ over variables $\mathbf{x}$, such that (1) $\phi \wedge \chi \not\models \text{false}$; and (2) $\phi \wedge \chi \models C$. A solution ϕ to the standard abduction problem is an interpretation of $R(\mathbf{x})$ that strengthens the left-hand side of the implication in order to make the implication logically valid. Every solvable abduction problem has an unique logically weakest solution. The procedure $\texttt{Abduce}(\chi, C, \mathbf{x})$ that solves the problem $R(\mathbf{x}) \wedge \chi \implies C$ is implemented in the EXPLAIN tool [8]. It computes the logically weakest solution containing the fewest number of variables. That is, it finds the most general and simple solution.

Proposition 2 ([8]). *If the abduction problem is solvable,* $\texttt{Abduce}(\chi, C, \mathbf{x})$ *terminates with an unique weakest solution containing a fewest number of variables.*

4 Synthesis Algorithm

In this section, we present our synthesis algorithm for solving the generalized sketching problem. In particular, we employ the abstract interpretation-based analyses, $\texttt{Inv}^\sharp_{d_{\mathcal{I}}}$ and $\texttt{Cond}^\sharp_{d_{\mathcal{F}}}$, as well as the logical abduction procedure, Abduce, to automatically find expressions for the holes in a program sketch, so that the resulting complete program satisfies its assertions.

Algorithm 1: `GenSketching`(p, H)

Input: Program sketch p, and a set of holes H
Output: Complete control function ϕ or an empty set

1 $\phi := \emptyset$;
2 **for** $(\texttt{??}_i \in H)$ **do**
3 $\quad ①s_i\,①' := \texttt{Extract}(p, \texttt{??}_i)$;
4 $\quad e_i := \texttt{Solve}(p, ①s_i\,①')$;
5 $\quad$ **if** $(e_i = \emptyset)$ **then return** $\emptyset$;
6 $\quad \phi := \phi \uplus [\texttt{??}_i \mapsto e_i]$
7 **for** $(\texttt{??}_i \in H)$ **do**
8 $\quad ①s_i\,①' := \texttt{Extract}(p, \texttt{??}_i)$;
9 $\quad \phi_i := \phi[\texttt{??}_i \mapsto \bot]$;
10 $\quad e_i' := \texttt{Solve}(p[\phi_i], ①s_i\,①')$;
11 $\quad$ **if** $(e_i = \emptyset)$ **then return** $\emptyset$;
12 $\quad \phi := \phi \uplus [\texttt{??}_i \mapsto e_i']$
13 **return** ϕ

High-level description. The `GenSketching`(p, H) synthesis procedure is shown in Algorithm 1. The procedure takes as input two parameters: a program sketch p, and a set of holes H in p. For each hole $\texttt{??}_i$ in H, we first find an initial solution, an expression e_i, where all other holes are treated as non-deterministic choices over integers or booleans (lines 2–6). This is achieved by identifying the statement "$①s_i\,①'$" in which $\texttt{??}_i$ occurs using $\texttt{Extract}(p, \texttt{??}_i)$, and by calling the function $\texttt{Solve}(p, ①s_i\,①')$ to find the expression e_i corresponding to hole $\texttt{??}_i$. This way, we construct an *initial* complete control function $\phi : [\texttt{??}_i \mapsto e_i]$ by using the above initial solutions for all holes in H. Then, we weaken the solution ϕ by iteratively weakening initial solutions for all holes (lines 7–12). To weaken the solution for each $\texttt{??}_i$, we fix the solutions e_j for all other $\texttt{??}_j$ in a partial control function ϕ_i, such that $\phi_i(\texttt{??}_i) = \bot$ and $\phi_i(\texttt{??}_j) = \phi(\texttt{??}_j)$ for all other $\texttt{??}_j \in H$. Next, we construct a program sketch $p[\phi_i]$ with only one hole $\texttt{??}_i$. Finally, we call $\texttt{Solve}(p[\phi_i], ①s_i\,①')$ to find the weaken expression e_i' for $\texttt{??}_i$. That is, we use the existing solution given by the current control function for all other holes and infer the weakest expression for $\texttt{??}_i$ that implies the assertion validity.

Solving one-hole sketches. The function $\texttt{Solve}(p, ①s\,①')$, shown in Algorithm 2, takes two parameters: a program sketch p and a statement "$①s\,①'$" in which the hole `??` we want to handle occurs. Note that all other holes in p are treated (analyzed) as non-deterministic choices: $[-\infty, \infty]$ for arithmetic and $\{\text{true}, \text{false}\}$ for boolean expressions. We first call a forward abstract analysis $\overrightarrow{[\![p]\!]}^\sharp d_{\mathcal{I}}$, where $d_{\mathcal{I}} = \top_{\mathbb{D}}$, to compute the invariants $\texttt{Inv}^\sharp_{d_{\mathcal{I}}}$ (line 1). Then we reason by the structure of the statement "$①s\,①'$". For assignments and `if`-s, we call a backward abstract analysis $\overleftarrow{[\![p]\!]}^\sharp d_{\mathcal{F}}$, where $d_{\mathcal{F}} = \texttt{FILTER}_{\mathbb{D}}(be^f, \texttt{Inv}^\sharp_{d_{\mathcal{I}}}(ⓕ))$, to compute the sufficient conditions $\texttt{Cond}^\sharp_{d_{\mathcal{F}}}$ of program p. When s is an assignment "x `=` `??`" (lines 2-7), we construct an abduction query where the premise is $\texttt{Inv}^\sharp_{d_{\mathcal{I}}}(①)[x'/x]$,

Algorithm 2: $\texttt{Solve}(p, \textcircled{l}s\,\textcircled{l'})$

Input: Program sketch p, and a statement $\textcircled{l}s\,\textcircled{l'}$ in which a hole occurs
Output: Expression e or an empty set

1 $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}} := \overrightarrow{[\![p]\!]}^{\sharp}\, d_{\mathcal{I}}$;
2 **switch** *(s)* **do**
3 **case** $(x := \texttt{??})$ **do**
4 $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}} := \overleftarrow{[\![p]\!]}^{\sharp}\, d_{\mathcal{F}}$;
5 $R(\mathbf{x}) := \texttt{Abduce}(\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(\textcircled{l}))[x'/x], \texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(\textcircled{l'}), \mathbf{x})$;
6 **if** *(*$\textit{unsat}(R(\boldsymbol{x}))$*)* **then return** $\emptyset$;
7 **return** $\texttt{SolveAsg}(R(\mathbf{x}), x)$
8 **case** $(\textit{if}\,(\texttt{??})\,\textcircled{t}s_1\,\textit{else}\,\textcircled{e}s_2)$ **do**
9 $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}} := \overleftarrow{[\![p]\!]}^{\sharp}\, d_{\mathcal{F}}$;
10 $R_{true}(\mathbf{x}) := \texttt{Abduce}(\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(\textcircled{l}), \texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(\textcircled{t}), \mathbf{x})$;
11 $R_{false}(\mathbf{x}) := \texttt{Abduce}(\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(\textcircled{l}), \texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(\textcircled{e}), \mathbf{x})$;
12 **if** *(*$\textit{unsat}(R_{true}(\boldsymbol{x})) \vee \textit{unsat}(R_{false}(\boldsymbol{x}))$*)* **then return** $\emptyset$;
13 **return** $\texttt{SolveCond}(R_{true}(\mathbf{x}), R_{false}(\mathbf{x}))$
14 **case** $(\textit{while}\,\textcircled{h}(\texttt{??}_i)\ \textit{do}\ \textcircled{b}s_1)$ **do**
15 $R_{false}(\mathbf{x}) := \texttt{Abduce}(\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(\textcircled{h}), d_{\mathcal{F}}, \mathbf{x})$;
16 $(\mathbf{x}{=}\mathbf{n}) = \texttt{Model}(R_{false}(\mathbf{x}))$;
17 $\texttt{Cond}^{\sharp,1}_{d_{\mathcal{F}}} = \overleftarrow{[\![p[\texttt{??}_i \mapsto (\mathbf{x} > \mathbf{n}) \wedge \texttt{??}_i]]\!]}^{\sharp}(d_{\mathcal{F}})$;
18 $\texttt{Cond}^{\sharp,2}_{d_{\mathcal{F}}} = \overleftarrow{[\![p[\texttt{??}_i \mapsto (\mathbf{x} < \mathbf{n}) \wedge \texttt{??}_i]]\!]}^{\sharp}(d_{\mathcal{F}})$;
19 $R^1_{true}(\mathbf{x}) := \texttt{Abduce}(\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(\textcircled{h}), \texttt{Cond}^{\sharp,1}_{d_{\mathcal{F}}}(\textcircled{b}), \mathbf{x})$;
20 $R^2_{true}(\mathbf{x}) := \texttt{Abduce}(\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(\textcircled{h}), \texttt{Cond}^{\sharp,2}_{d_{\mathcal{F}}}(\textcircled{b}), \mathbf{x})$;
21 **if** *(*$\textit{unsat}(R_1(\boldsymbol{x})) \vee \textit{unsat}(R_2(\boldsymbol{x}))$*)* **then return** $\emptyset$;
22 **return** $(\mathbf{x} > \mathbf{n} \wedge R^1_{true}(\mathbf{x})) \vee (\mathbf{x} < \mathbf{n} \wedge R^2_{true}(\mathbf{x}))$

the desired conclusion is $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(\textcircled{l'})$, and the unknown predicate (abducible) $R(\mathbf{x})$ is defined over all variables $\mathbf{x}$ that are in scope of s_i including x and x'. We denote by $[x'/x]$ the renaming of x as x'. Moreover, we configure the call to `Abduce` so that the variable x has the highest priority to occur in the solution $R(\mathbf{x})$ of the given abduction query. Then we call $\texttt{SolveAsg}(R(\mathbf{x}))$ to find one expression e_i such that $x = e_i$ satisfies the predicate $R(\mathbf{x})$. This is realized by asking an SMT solver for one interpretation (model) of the formula $R(\mathbf{x})$. Finally, we return the expression $e_i[x/x']$ as solution of this case. We recall the example `intro.c` in Section 2 to see how this case works in practice.

When s is a conditional statement "`if (??) then` $\textcircled{t}s_1$ `else` $\textcircled{e}s_2$" (lines 8–13), we construct two abduction queries in which the premise is $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(\textcircled{l})$ and the unknown predicate is defined over all variables $\mathbf{x}$ that are in scope of s. The conclusions are $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(\textcircled{t})$ and $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(\textcircled{e})$ in the first and second abduction query, respectively. We then call $\texttt{SolveCond}(R_{true}(\mathbf{x}), R_{false}(\mathbf{x}))$ to check if $\neg R_{true}(\mathbf{x}) \Longrightarrow R_{false}(\mathbf{x})$ by an SMT solver, where $R_{true}(\mathbf{x})$ and $R_{false}(\mathbf{x})$ are

solutions of the first and second abduction query. If this is true, then $R_{true}(\mathbf{x})$ is returned as solution for this case. Otherwise, $R_{true}(\mathbf{x})$ is strengthen until $\neg R_{true}(\mathbf{x}) \Longrightarrow R_{false}(\mathbf{x})$ holds, in which case the found $R_{true}(\mathbf{x})$ is returned as solution. For example, see how `abs.c` in Section 2 is resolved.

Similarly, we handle the case when s is an iteration "`while` ⓗ`(??) do` ⓑs_1" (lines 14-22). First, we construct an abduction query where the premise is $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓗ)$, the desired conclusion is $d_{\mathcal{F}}$, and the unknown predicate (abducible) is $R_{false}(\mathbf{x})$. Then, we find one interpretation ($\mathbf{x}{=}\mathbf{n}$) of the formula $R_{false}(\mathbf{x})$, which is obtained by finding a model M of $R_{false}(\mathbf{x})$ by an SMT solver and setting $\mathbf{x} = M(\mathbf{x})$. Next, we perform two backward analyses defined as follows: $\texttt{Cond}^{\sharp,1}_{d_{\mathcal{F}}} = \overleftarrow{[\![p[\texttt{??}_i \mapsto (\mathbf{x} > \mathbf{n}) \wedge \texttt{??}_i]]\!]^{\sharp}}(d_{\mathcal{F}})$ and $\texttt{Cond}^{\sharp,2}_{d_{\mathcal{F}}} = \overleftarrow{[\![p[\texttt{??}_i \mapsto (\mathbf{x} < \mathbf{n}) \wedge \texttt{??}_i]]\!]^{\sharp}}(d_{\mathcal{F}})$. We create two abduction queries using: (1) $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓗ)$, $\texttt{Cond}^{\sharp,1}_{d_{\mathcal{F}}}(ⓑ)$, $R^1_{true}(\mathbf{x})$; and (2) $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓗ)$, $\texttt{Cond}^{\sharp,2}_{d_{\mathcal{F}}}(ⓑ)$, and $R^2_{true}(\mathbf{x})$. Finally, the solution is $(\mathbf{x} > \mathbf{n} \wedge R^1_{true}(\mathbf{x})) \vee (\mathbf{x} < \mathbf{n} \wedge R^2_{true}(\mathbf{x}))$. For example, see how `while.c` in Section 2 works.

Correctness. The following theorem states correctness of the `GenSketching` algorithm.

Theorem 1. *`GenSketching`(p, H) is correct and terminates.*

Proof. The procedure `GenSketching`(p, H) terminates since all steps in it are terminating. The correctness of `GenSketching`(p, H) follows from the soundness of $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}$ and $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}$ (see Proposition 1) and the correctness of `Abduce` (see Proposition 2).

The correctness proof is by structural induction on statements s in programs p of the form "ⓘ s; ⓕ `assert` (be^f)". We consider the case of assignment `x=??`. Since there is one hole in p, we call `Solve`$(p,$ ⓘ`x=??` ⓕ$)$. We infer $\texttt{Inv}^{\sharp}_{d_{\mathcal{I}}}(ⓘ) = \top_{\mathbb{D}}$ and $\texttt{Cond}^{\sharp}_{d_{\mathcal{F}}}(ⓕ) = be^f$, so we obtain the abduction query `Abduce`$(\text{true}, be^f, \mathbf{x})$. The solution is $R(\mathbf{x}) \equiv be^f$, thus we call `SolveAsg`$(be^f, \mathbf{x})$ to find one expression ae such that `x=`ae satisfies be^f. By construction of ae, it follows that the program "ⓘ `x=`ae; ⓕ `assert` (be^f)" is valid. Similarly, we handle the other cases. □

5 Evaluation

We now evaluate our approach for generalized program sketching. The evaluation aims to show that we can use our approach to efficiently resolve various C program sketches with numerical data types.

Implementation We have implemented our synthesis algorithm in a prototype tool. The abstract operations and transfer functions of the numerical abstract domains (e.g. Polyhedra [7]) are provided by the APRON library [25], while the abduction and SMT queries are solved by the EXPLAIN [8] and the MISTRAL [9] tools. Our tool is written in OCaml and consists of around 7K lines of code. Currently, it provides only a limited support for arrays, pointers, `struct` and `union` types.

```
void main(int n1, int n2, int n3){
① int max = n1;
② if (n2>max) max = n2;
③ if (n3>max) max = ??;
// if (??) max = n3;
④ assert (max ≥ n1);
⑤ assert (max ≥ n2);
⑥ assert (max ≥ n3); }
```

Fig. 8. `max.c`.

```
void main(int n){
① int x = n;
② if (??) x = x+2n;
③ else x = x-2n;
④ assert (x ≤ 3); }
```

Fig. 9. `cond.c`.

```
void main() {
① unsigned int j;
② int i = 0;
③ j = [0,9]; //j = ??;
④ while (??) {
  //while (i<100) {
⑤   i = i+1;
⑥   j = j+[0,1]; }
⑦ assert (j ≤ 105); }
```

Fig. 10. `mine.c`.

Experiment setup and Benchmarks All experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU, Lubuntu VM, with 8 GB memory, and we use a time-out value of 300 sec. All times are reported as average over five independent executions. We report times needed for the actual synthesis task to be performed. The implementation, benchmarks, and all obtained results are available from [16]: https://zenodo.org/record/8165119. We compare our approach `GenSketching` based on the Polyhedra domain with program sketching tool SKETCH version 1.7.6 that uses SAT-based inductive synthesis [33,32], as well as with the FAMILYSKETCHER that uses lifted (family-based) static analysis by abstract interpretation (the Polyhedra domain) [19]. Note that SKETCH and FAMILYSKETCHER can only solve the standard sketching problem, where the unknown holes represent some integer constants. Therefore, they cannot resolve our benchmarks. We need to do some simplifications, so that the unknown holes refer to integer constants. Moreover, their synthesis times depend on the sizes of hole domains (and inputs for SKETCH). Hence, for SKETCH and FAMILYSKETCHER we report synthesis times to resolve simplified sketches with `5-bits` and `10-bits` sizes of unknown holes. On the other hand, `GenSketching` can synthesize arbitrary expressions and its synthesis time does not depend on the sizes of holes. For `GenSketching`, we report TIME which is the total time to resolve a given sketch, and ABDTIME which is the time to solve the abduction queries in the given synthesis task.

The evaluation is performed on several C numerical sketches collected from the SKETCH project [33,32], SV-COMP (https://sv-comp.sosy-lab.org/), and the literature [28]. In particular, we use the following benchmarks: `intro.c` (Fig. 1), `abs.c` (Fig. 2), `while.c` (Fig. 3), `max.c` (Fig. 8), `cond.c` (Fig. 9), and `mine.c` (Fig. 10).

Performance Results Table 1 shows the performance results of synthesizing our benchmarks. To handle `intro.c` using SKETCH and FAMILYSKETCHER, we need to simplify it so that the hole represents an integer constant. This is done by replacing `y = ??` by `y = y-??` or `y = y+??`. Moreover, they cannot handle non-deterministic choices, so we use constants instead. SKETCH and FAMILYSKETCHER resolve the simplified `intro.c` in 0.208 sec and 0.0013 sec for 5-bit sizes of holes, while `GenSketching` resolves the `intro.c` in 0.0022 sec.

Table 1. Performance results of `GenSketching` vs. SKETCH vs. FAMILYSKETCHER. All times in sec.

Bench.	`GenSketching`		SKETCH		FAMILYSKETCHER	
	TIME	ABDTIME	`5-bits`	`10-bits`	`5-bits`	`10-bits`
`intro.c`	0.0022	0.0011	0.208	0.239	0.0013	0.0016
`abs.c`	0.0021	0.0013	0.204	0.236	0.0020	0.0021
`while.c`	0.0055	0.0013	0.213	0.224	0.0047	0.0053
`max.c`	0.0042	0.0011	0.229	24.25	0.0025	0.0026
`max2.c`	0.0040	0.0015	0.227	31.88	0.0022	0.0033
`cond`	0.0019	0.0011	1.216	2.362	0.0021	0.0023
`mine.c`	0.0757	0.0012	0.236	1.221	0.0035	0.0042
`mine2.c`	0.0059	0.0013	0.215	1.217	0.0024	0.0031

To resolve `abs.c` using SKETCH and FAMILYSKETCHER, we use the `if`-guard ($\texttt{n} \leq \texttt{??}$). In this case SKETCH and FAMILYSKETCHER terminate in 0.204 sec and 0.0022 sec for 5-bit sizes of holes, while `GenSketching` terminates in 0.0021 sec. Still, FAMILYSKETCHER reports "I don't know" answer due to the precision loss. In particular, it infers the invariant ($\texttt{2*??-n+abs} \geq 0 \wedge 0 \leq \texttt{??} \leq 31 \wedge \texttt{n+abs} \geq 0$) at loc. ④, thus it is unable to conclude for which values of `??` the assertion ($\texttt{abs} \geq 0$) will hold. For similar reasons, FAMILYSKETCHER cannot successfully resolve the other simplified sketches that contain holes in `if`-guards (see `max2.c` and `cond.c` below).

We simplify the `while`-guard of `while.c` to ($\texttt{x>??}$). SKETCH still cannot resolve this example, since it uses only 8 unrollments of the loop by default. If the loop is unrolled 10 times, SKETCH terminates in 0.213 sec for 5-bit sizes of holes. FAMILYSKETCHER terminates in 0.0047 sec for 5-bits, while `GenSketching` terminates in 0.0055 sec.

Consider the program sketch `max.c` in Fig. 8 that finds a maximum of three integers. It contains one hole in the assignment at loc. ③. The forward analysis of `GenSketching` generates the invariant ($\texttt{max}' \geq \texttt{n1} \wedge \texttt{max}' \geq \texttt{n2} \wedge \texttt{n3} \geq \texttt{max}'$) before the hole (where $\texttt{max}'$ denotes the value of `max` before the hole), while the backward analysis infers the sufficient condition ($\texttt{max} \geq \texttt{n1} \wedge \texttt{max} \geq \texttt{n2} \wedge \texttt{max} \geq \texttt{n3}$) after the hole. The result of the corresponding abduction query is ($\texttt{max} = \texttt{n3}$), so we fill the hole with the assignment `max = n3`. For SKETCH and FAMILYSKETCHER, we simplify the hole to `max = n3-??`. Since there are three inputs and one hole, SKETCH takes 24.25 sec to resolve this simplified sketch for 10-bit sizes and timeouts for bigger sizes.

Consider a variant of `max.c`, denoted `max2.c`, where the commented statement in Fig. 8 is placed at loc. ③, so the hole is `if`-guard. The forward analysis of `GenSketching` infers the invariant ($\texttt{max} \geq \texttt{n1} \wedge \texttt{max} \geq \texttt{n2}$) before the hole, while the backward analysis infers the sufficient conditions ($\texttt{n3} \geq \texttt{n1} \wedge \texttt{n3} \geq \texttt{n2}$) at the `then` branch after the hole and ($\texttt{max} \geq \texttt{n1} \wedge \texttt{max} \geq \texttt{n2} \wedge \texttt{max} \geq \texttt{n3}$) at the `else` branch after the hole. We construct two abduction queries, and the results

are ($\texttt{max} \leq \texttt{n3}$) and ($\texttt{max} \geq \texttt{n3}$), respectively. Hence, we fill the hole with the boolean expression ($\texttt{max} \leq \texttt{n3}$). For SKETCH and FAMILYSKETCHER, we use the simplified sketch where the `if`-guard is (`max<??`). SKETCH timeouts for bigger than 10-bit sizes of holes and inputs, while FAMILYSKETCHER returns "I don't know" answer.

The sketch `cond.c` contains an `if`-guard hole. The inferred invariant of `GenSketching` is (`x=n`) before the hole, while sufficient conditions are ($\texttt{x+2n} \leq 3$) at the `then` branch and ($\texttt{x-2n} \leq 3$) at the `else` branch after the hole. The results of the two abductions queries are $R_{true}(\texttt{x,n}) \equiv (\texttt{n} \leq 1)$ and $R_{false}(\texttt{x,n}) \equiv (\texttt{n>-4})$. Since $(\neg \texttt{n} \leq 1) \implies (\texttt{n>-4})$ is valid, the hole is filled with $(\texttt{n} \leq 1)$.

Consider the `mine.c` sketch, taken from [28], containing a `while`-guard and the `GenSketching` approach. The forward analysis infers the `while`-invariant ($\texttt{j} \geq 0 \wedge \texttt{i} \geq 0 \wedge \texttt{i} \geq \texttt{j-9}$). Hence, the answer to the first abduction query $(\texttt{j} \geq 0 \wedge \texttt{i} \geq 0 \wedge \texttt{i} \geq \texttt{j-9}) \wedge R_{false}(i,j) \implies (\texttt{j} \leq 105)$ is $(\texttt{i} \leq 96)$. One solution is $(\texttt{i} = 96)$, so we perform two backward analyses with `while`-guards ($\texttt{i} < 96 \wedge \texttt{??}$) and ($\texttt{i} > 96 \wedge \texttt{??}$). We construct two abduction queries and obtain $R^1_{true}(i,j) \equiv$ (true) and $R^2_{true}(i,j) \equiv$ (false). Thus, we fill the hole with $(\texttt{i} < 96)$.

Consider a variant of `mine.c`, denoted `mine2.c`, where the commented statements (see Fig. 10) are placed at locs. ③ and ④, so the hole is in the assignment `j = ??`. The invariant at loc. ③ is (`i=0`), while the sufficient condition obtained at loc. ④ is ($\texttt{j} \geq 0 \wedge \texttt{i} \geq 0 \wedge \texttt{i} \geq \texttt{j-5} \wedge \texttt{j} \leq 105$). Note that the backward sufficient condition analysis infers conditions at all locations so that all executions branching from those locations will satisfy the assertion. By solving the corresponding abduction query, we obtain the answer $(0 \leq \texttt{j} \leq 5)$, so we fill the hole at loc. ③ with $\texttt{j} = [0,5]$. Even if the non-deterministic choice $[0,1]$ always evaluates to 1 in the `while`-body, the assertion $(\texttt{j} \leq 105)$ will hold in the resulting complete program where the initial value of `j` is in $[0,5]$. For SKETCH and FAMILYSKETCHER to successfully handle the simplified versions of `mine.c` and `mine2.c` we need to use constants instead of non-deterministic choices.

Discussion. In summary, we can conclude that `GenSketching` can successfully synthesize the holes in all sketches, and it does not depend on the sizes of holes thus outperforming the other tools. The abduction time is proportional to the number of abduction queries that need to be solved in a given synthesis task. FAMILYSKETCHER achieves comparable synthesis times to `GenSketching`. FAMILYSKETCHER performs one forward (lifted) analysis using decision tree abstract elements, in which decision nodes are linear constraints defined over unknown holes and the leaves provide analysis information (in the form of polyhedral constraints) corresponding to each partition of the tree. Furthermore, FAMILYSKETCHER has similar synthesis times for 5-bit and 10-bit sizes of holes, since all integer holes in the given simplified examples can be handled symbolically. However, if a hole occurs in more complex expressions, e.g. `??*x+y`, then the performance of FAMILYSKETCHER will decline since the hole should be handled explicitly [19].

The current tool supports an interesting subset of C, so we can handle many interesting benchmarks. The selected benchmarks represent the proof-of-concept

that our approach can be successfully applied in practice. They are chosen to show some distinctive features of our approach compared to the other state-of-the-art tools. To handle bigger programs, we can use some computationally cheaper abstract domains, such as octagons and intervals, but this will result in additional precision loss that will influence the precision of our approach. For example, if we use intervals domain we cannot handle `abs.c` example from Section 2.

6 Related Work

Abstract interpretation in program analysis and verification. Forward invariance and backward sufficient condition analyses by abstract interpretation have been used in practice for a long time [5,7,29,3,28]. The three most popular forward invariance analyses are based on the Interval [5], the Octagon [27], and the Polyhedra [7] domains that infer variable bounds, unit affine inequalities and arbitrary affine inequalities on variables, respectively. Sufficient condition analysis has been first introduced by Bourdoncle [3] in his work on abstract debugging of deterministic programs. Miné [28] has presented a technique for automatically inferring sufficient conditions of non-deterministic programs by using a polyhedral backward analysis. The under-approximating sound abstract operators are implemented as part of the APRON library in the Banal tool.

Several works use a combination of forward-backward analyses to extract interesting dynamic properties of programs [3,21,30,34]. In particular, the combination of forward-backward analyses are used by Rival [30] to obtain a set of traces that may lead to error; by Bourdoncle [3] to find preconditions for invariant and intermittent assertions to always hold; by Dimovski and Legay [21] to calculate the probability that a target assertion is satisfied/violated; and by Urban and Miné [34] to infer ranking functions for proving termination.

Many recent approaches and tools for program verification use static analysis by abstract interpretation. ASTREE [6] is an industrial-scale static analyzer for verifying avionics software. SeaHorn [23] combines Horn-clause solving techniques with abstract interpretation, Pagai [24] combines SMT-solving with abstract interpretation, whereas Ulitimate Taipan [22] is a CEGAR-based software model checker that uses abstract interpretation to derive invariants for the path program corresponding to a given spurious counterexample. VATer [35] uses input space partitioning to iteratively refine static analyses by abstract interpretation, and moreover it uses bounded exhaustive testing to complement static analyses.

Program sketching. The goal of the so-called standard sketching problem is to synthesize program sketches in which some missing holes are integer constants. One of the earliest and widely-known approach to solve the standard sketching problem is the Sketch tool [32,33], which uses SAT-based counterexample-guided inductive synthesis. It iteratively generates a finite set of inputs and performs SAT queries to identify values for the holes so that the resulting program satisfies all assertions for the given inputs. Further SAT queries are then

used to check if the resulting program is correct on all possible inputs. Hence, SKETCH may need several iterations to converge reporting only one solution. Moreover, SKETCH reasons about loops by unrolling them, so is very sensitive to the degree of unrolling. Our approach does not have this constraint, and is able to handle unbounded loops in a sound way. Still, our approach can be applied to numerical programs, while SKETCH is more general and especially suited for bit-manipulating programs. Another works for solving this standard sketching problem are proposed in [19,15,17], which use a lifted (family-based) static analysis by abstract interpretation [18,13,14,20]. The key idea is that the set of all possible sketch realizations represent a program family with numerical features. Thus, the effort of conducting an effective search of all possible hole realizations is delegated to an efficient lifted static analyzer for program families, which uses a specifically designed decision tree abstract domain. However, the above works address the standard sketching problem, where each unknown hole can be replaced by one value from a finite set of integers. In this work, we pursue this line of work by considering the generalized sketching problem, where each hole can be replaced by an arbitrary expression. This way, we broaden the space of program sketches that can be resolved.

Recently, abstract interpretation has been successfully applied to program synthesis [31,36]. The work [31] efficiently synthesizes imperative programs from input-output examples. It combines the enumerative search with static analysis, which is used to identify and ignore partial programs that fail to be a solution. The work [36] uses specifications given as input-output examples and a bitvector abstract domain to synthesize bitvector-manipulating programs without loops. We could replace the Polyhedra domain with the bitvector domain from [36] in our approach in order to scale to bigger programs. However, the abduction solver EXPLAIN currently supports only the linear arithmetic SMT theory (not other theories, such as bitvector). If an abduction solver that supports bitvecotrs becomes available in the future, we can extend our approach to handle bitvector-manipulating programs as well.

Abduction in program analysis. Logical abduction has found a number of applications in program analysis. In the context of separation logic for shape analysis, abduction (or bi-abduction) are used for performing modular heap reasoning [4]. An abduction algorithm for first-order SMT theories is described in [8] for computing a maximally simple and general solution. This form of SMT-based abduction has also been applied for loop invariant generation [11], for error explanation and diagnosis of error reports generated by verification tools [10], and for construction of circular compositional program proofs [26].

Abduction in program synthesis. The abduction has been used before in program synthesis to infer missing guards from low-level C code such that all buffer accesses are memory safe [12]. However, this is the constraint-based approach that uses Hoare logic-style verification condition (VC) generation with a logical abduction algorithm to solve the generated constraints. To generate verification conditions, the approach computes the strongest postcondition before the missing guard,

as well as the weakest precondition that guarantees memory safety after the missing guard. Since this approach uses loop invariants provided by an external tool in the VC generation phase, the quality of the solutions is relative to the used loop invariants. Unlike the prior constraint-based approach to synthesis [12], our method uses exclusively abstract interpretation-based techniques to infer invariant postconditions before the hole and sufficient preconditions after the hole. Hence, no assist from external tools is used. Moreover, our approach can synthesize arbitrary expressions in general C programs, whereas the prior approach [12] can only synthesize boolean guards in `if`-s that ensure memory safety in buffer accessing C programs.

7 Conclusion

In this paper, we present an approach for program synthesis by interaction between abstract interpretation and logical abduction. We introduce a synthesis algorithm that infers arbitrary expressions for unknown holes in program sketches, so that the resulting complete programs satisfy all assertions. We experimentally demonstrate the effectiveness of our approach for generating correct solutions of a variety of C benchmarks.

In the future, we would like to extend our approach for program synthesis by considering program sketches in which apart from expressions, unknown holes can be arbitrary statements as well. We hope further fruitful interplay between abstract interpretation and logical abduction will be possible.

References

1. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016. pp. 789–801. ACM (2016). https://doi.org/10.1145/2837614.2837628, https://doi.org/10.1145/2837614.2837628
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013. pp. 1–8. IEEE (2013), http://ieeexplore.ieee.org/document/6679385/
3. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI). pp. 46–55. ACM (1993). https://doi.org/10.1145/155090.155095, https://doi.org/10.1145/155090.155095
4. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009. pp. 289–300. ACM (2009). https://doi.org/10.1145/1480881.1480917, https://doi.org/10.1145/1480881.1480917
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Record of the Fourth ACM Symposium on POPL. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, http://doi.acm.org/10.1145/512950.512973

6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings. LNCS, vol. 3444, pp. 21–30. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_3, https://doi.org/10.1007/978-3-540-31987-0_3
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conference Record of the Fifth Annual ACM Symposium on POPL'78. pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770, https://doi.org/10.1145/512760.512770
8. Dillig, I., Dillig, T.: Explain: A tool for performing abductive inference. In: Computer Aided Verification - 25th International Conference, CAV 2013. Proceedings. LNCS, vol. 8044, pp. 684–689. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_46, https://doi.org/10.1007/978-3-642-39799-8_46
9. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings. LNCS, vol. 5643, pp. 233–247. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_20, https://doi.org/10.1007/978-3-642-02658-4_20
10. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12. pp. 181–192. ACM (2012). https://doi.org/10.1145/2254064.2254087, https://doi.org/10.1145/2254064.2254087
11. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013. pp. 443–456. ACM (2013). https://doi.org/10.1145/2509136.2509511, https://doi.org/10.1145/2509136.2509511
12. Dillig, T., Dillig, I., Chaudhuri, S.: Optimal guard synthesis for memory safety. In: Computer Aided Verification - 26th International Conference, CAV 2014. Proceedings. LNCS, vol. 8559, pp. 491–507. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_32, https://doi.org/10.1007/978-3-319-08867-9_32
13. Dimovski, A.S.: A binary decision diagram lifted domain for analyzing program families. J. Comput. Lang. **63**, 101032 (2021). https://doi.org/10.1016/j.cola.2021.101032, https://doi.org/10.1016/j.cola.2021.101032
14. Dimovski, A.S.: Lifted termination analysis by abstract interpretation and its applications. In: GPCE '21: Concepts and Experiences, 2021. pp. 96–109. ACM (2021). https://doi.org/10.1145/3486609.3487202, https://doi.org/10.1145/3486609.3487202
15. Dimovski, A.S.: Quantitative program sketching using lifted static analysis. In: Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Proceedings. LNCS, vol. 13241, pp. 102–122. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_6, https://doi.org/10.1007/978-3-030-99429-7_6
16. Dimovski, A.S.: Artifact for the paper "generalized program sketching by abstract interpretation and logical abduction". Zenodo (2023). https://doi.org/10.5281/zenodo.8165119, https://doi.org/10.5281/zenodo.8165119
17. Dimovski, A.S.: Quantitative program sketching using decision tree-based lifted analysis. J. Comput. Lang. **75**, 101206 (2023). https://doi.org/10.1016/j.cola.2023.101206, https://doi.org/10.1016/j.cola.2023.101206

18. Dimovski, A.S., Apel, S.: Lifted static analysis of dynamic program families by abstract interpretation. In: 35th European Conference on Object-Oriented Programming, ECOOP 2021. LIPIcs, vol. 194, pp. 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.14, https://doi.org/10.4230/LIPIcs.ECOOP.2021.14
19. Dimovski, A.S., Apel, S., Legay, A.: Program sketching using lifted analysis for numerical program families. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings. LNCS, vol. 12673, pp. 95–112. Springer (2021). https://doi.org/10.1007/978-3-030-76384-8_7, https://doi.org/10.1007/978-3-030-76384-8_7
20. Dimovski, A.S., Apel, S., Legay, A.: Several lifted abstract domains for static analysis of numerical program families. Sci. Comput. Program. **213**, 102725 (2022). https://doi.org/10.1016/j.scico.2021.102725, https://doi.org/10.1016/j.scico.2021.102725
21. Dimovski, A.S., Legay, A.: Computing program reliability using forward-backward precondition analysis and model counting. In: Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Proceedings. LNCS, vol. 12076, pp. 182–202. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_9, https://doi.org/10.1007/978-3-030-45234-6_9
22. Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate taipan: Trace abstraction and abstract interpretation - (competition contribution). In: 23rd International Conference, TACAS 2017, Proceedings, Part II. LNCS, vol. 10206, pp. 399–403 (2017). https://doi.org/10.1007/978-3-662-54580-5_31, https://doi.org/10.1007/978-3-662-54580-5_31
23. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I. LNCS, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20, https://doi.org/10.1007/978-3-319-21690-4_20
24. Henry, J., Monniaux, D., Moy, M.: PAGAI: A path sensitive static analyser. Electron. Notes Theor. Comput. Sci. **289**, 15–25 (2012). https://doi.org/10.1016/j.entcs.2012.11.003, https://doi.org/10.1016/j.entcs.2012.11.003
25. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings. LNCS, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52, https://doi.org/10.1007/978-3-642-02658-4_52
26. Li, B., Dillig, I., Dillig, T., McMillan, K.L., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. In: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013. Proceedings. LNCS, vol. 7795, pp. 370–384. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_26, https://doi.org/10.1007/978-3-642-36742-7_26
27. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1, https://doi.org/10.1007/s10990-006-8609-1
28. Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Sci. Comput. Program. **93**, 154–182 (2014). https://doi.org/10.1016/j.scico.2013.09.014, https://doi.org/10.1016/j.scico.2013.09.014

29. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages **4**(3-4), 120–372 (2017). https://doi.org/10.1561/2500000034, https://doi.org/10.1561/2500000034
30. Rival, X.: Understanding the origin of alarms in astrée. In: Static Analysis, 12th International Symposium, SAS 2005, Proceedings. LNCS, vol. 3672, pp. 303–319. Springer (2005). https://doi.org/10.1007/11547662_21, https://doi.org/10.1007/11547662_21
31. So, S., Oh, H.: Synthesizing imperative programs from examples guided by static analysis. In: Static Analysis - 24th International Symposium, SAS 2017, Proceedings. LNCS, vol. 10422, pp. 364–381. Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_18, https://doi.org/10.1007/978-3-319-66706-5_18
32. Solar-Lezama, A.: Program sketching. STTT **15**(5-6), 475–495 (2013). https://doi.org/10.1007/s10009-012-0249-7, https://doi.org/10.1007/s10009-012-0249-7
33. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation. pp. 281–294. ACM (2005). https://doi.org/10.1145/1065010.1065045, https://doi.org/10.1145/1065010.1065045
34. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Static Analysis - 21st International Symposium, SAS 2014. Proceedings. LNCS, vol. 8723, pp. 302–318. Springer (2014). https://doi.org/10.1007/978-3-319-10936-7_19, https://doi.org/10.1007/978-3-319-10936-7_19
35. Yin, B., Chen, L., Liu, J., Wang, J., Cousot, P.: Verifying numerical programs via iterative abstract testing. In: Static Analysis - 26th International Symposium, SAS 2019, Proceedings. LNCS, vol. 11822, pp. 247–267. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_13, https://doi.org/10.1007/978-3-030-32304-2_13
36. Yoon, Y., Lee, W., Yi, K.: Inductive program synthesis via iterative forward-backward abstract interpretation. In: PLDI '23: 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2023. p. 1657–1681. ACM (2023). https://doi.org/10.1145/3591288, https://doi.org/10.1145/3591288