Abstract Interpretation using a Language of Symbolic Approximation

Matthieu Lemerre, Sébastien Bardin

Abstract Interpretation using a Language of Symbolic Approximation

Matthieu Lemerre and Sébastien Bardin

CEA, LIST

Abstract. The traditional abstract domain framework for imperative programs suffers from several shortcomings; in particular it does not allow precise symbolic abstractions. To solve these problems, we propose a new abstract interpretation framework, based on symbolic expressions used both as an *abstraction* of the program, and as the *input* analyzed by abstract domains. We demonstrate new applications of the framework: an abstract domain that efficiently propagates constraints *across the whole program*; a new formalization of functor domains as *approximate translation*, which allows the production of *approximate programs*, on which we can perform classical symbolic techniques. We used these to build a complete analyzer for embedded C programs, that demonstrates the practical applicability of the framework.

1 Introduction

Context The usual *lattice-based* structuring of abstract interpreters for imperative programs by Cousot and Cousot [7] consists in associating, to each program point, the element of a lattice representing the over-approximation of the set of possible states at that point. The interface of abstract domains then consists in transfer functions that interpret the syntax of the program to compute the lattice element corresponding to the next program point; as well as lattice operations such as inclusion ($\sqsubseteq$), join ($\sqcup$), and widening (∇). This approach has been highly successful in both laying the theoretical foundation of software analysis techniques, and applying them in industrial applications [3, 12, 20].

Problem Despite these successes the design of industrial-strength analyzers is still a challenging art. We highlight two problems, detailed in Section 2.

First, the abstract state at each program point contains a mapping of the whole memory; yet the abstract state between nearby program points is nearly identical. Consequences include a high memory consumption and expensive operations. For instance, memory join is costly, because every memory location is joined, instead of only the few that differ. Mitigating this problem requires implementation tricks such as using functional data structures [2], possibly with hash-consing [20]; essentially the implementation tries to share what was duplicated by the theoretical framework.

Another important limitation is the handling of symbolic relations. Symbolic abstract domains are necessary in practical lattice-based static analyzers [13, 25,

4, 11, 22, 18], notably to handle the case where an intermediate computation is put in a temporary variable (like `cond` in Figure 1). These abstract domains are limited by the fact that they must fit in the lattice-based framework; in particular, their join and widening operations are always very imprecise.

Goal, challenge and proposal Because these limitations are hard to fix in the standard *lattice-based* framework, our goal is to design an alternative framework that fixes these issues; but also encompasses the lattice-based framework, so as to reuse previous work.

We propose a new *term-based* framework, based on a language of symbolic expressions (Section 3); which can be used both as the abstract state inside the abstract domain (i.e. replacing the lattice elements) or as the input of the abstract domain (e.g. replacing the program itself). Abstract domains are functions, that evaluate a symbolic expression to an *abstract value*, that can be *concretized* into a set of possible values for the variables of the expression (Section 4).

Contributions Our main contribution is the design of a new term-based abstract interpretation framework. Key ingredients include the Language of Approximation and Fixpoint (LAF) logic (Section 3) with *nondet* and μ operators – replacing join and widening; its collecting semantics; and the definition of abstract domains as abstract interpreter of LAF terms (Section 4.1).

Our second contribution is specific instances of term-based abstract domains: we show how to lift lattice-based abstract domains to term-based abstract domains (Section 4.2) (and demonstrate improved complexity in the case of non-relational abstract domains); we provide new abstract domains based on term rewriting (Section 4.3). In Section 5 we combine term rewriting and lattice-based abstract interpretation in a new domain, that performs backward and forward constraint propagation across the whole program.

Our last contribution is the evaluation of an early implementation of the approach (Section 6), but that already works on industrial case studies and large SVComp benchmarks. Term-based abstract interpretation allows to decompose the analyzer as a succession of transformation over LAF terms, each individually simple, but that mutually refine one another to provide a precise result. A first experiment demonstrates the interest of whole-program constraint propagation. A second shows how we leverage the production of approximate LAF terms by the term-based abstract domains, to export simplified formula to a Horn-based model checker.

2 Motivation and key ideas

Our method can be summarized as using a special language of symbolic expressions; used both as the abstract state inside an abstract domain, or as the program that the abstract domain analyze. We first cover the challenges of maintaining a precise symbolic abstraction, and why we require a new framework for doing so; the benefits of performing abstract interpretation over these symbolic expressions; and the organization of an analyzer structured using "translator abstract domains", both inputting and outputting symbolic expressions.

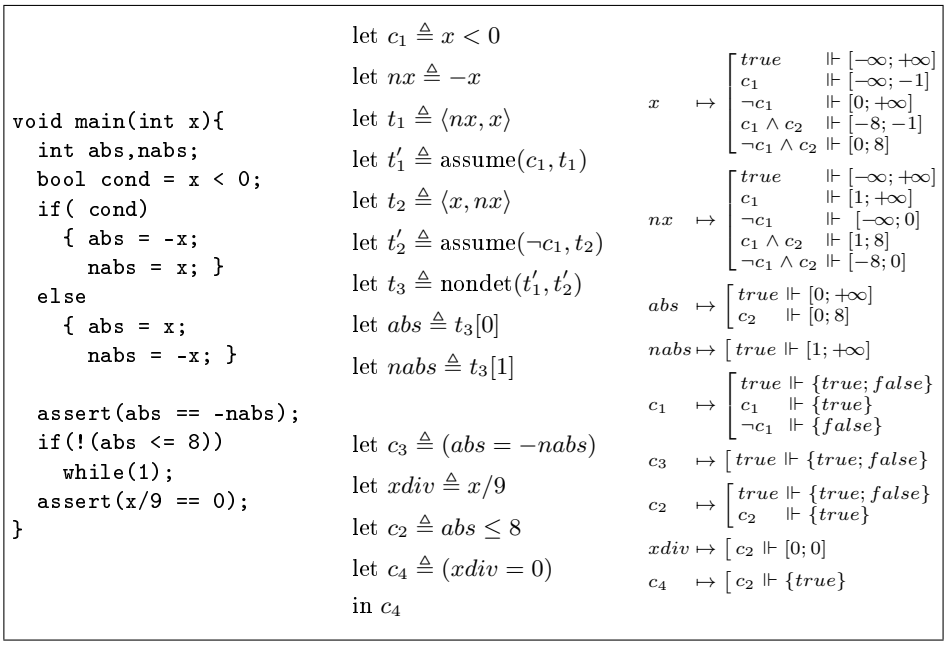

Fig. 1: Applying the constraint propagation abstract domain on a C program (left). Middle: term abstraction built by the domain; x represents the value of `x`. Right: mapping from variables to abstract values, according to some conditions.

Figure 1 illustrates the use of our language as an abstraction of the program (in the middle), and as the language on which abstract interpretation is performed (on the right). This example is explained in detail in Section 5.

2.1 Computing symbolic abstractions

Symbolic abstract domains compute abstractions represented by a *term* (expression) in some language. Existing symbolic abstract domains lose precision when assignment, join or widening is performed [25, 13, 11]. Indeed, having a fully precise symbolic abstract domain cause major difficulties, that cannot be solved using the standard interface to abstract domains.

Variables for values, not for locations In usual A.I. of imperative programs, a precision loss may occur when a variable `x` is overwritten: relations such as $y = 2 * x$ have to be "killed" (if x cannot be substituted). This affects relational abstract domains [19], including symbolic domains [25, 13, 11].

But this issue only occurs because variables represent (mutable) memory *locations*. When variables represent (immutable) *values*, formula remain valid regardless of which memory cell gets overwritten. In LAF (but also in [4, 5]), *variables denote values*, thus there is no need to kill any relation.

Symbolic join is not a least upper bound Usual A.I. requires abstract domains to have a lattice structure. We believe that this requirement does not fit well symbolic abstraction, and explain why below.

Intuitively, two expressions e_1 and e_2 can be "joined" using a new expression "nondet(e_1, e_2)", representing a non-deterministic choice between e_1 and e_2. But to avoid loosing precision, the resulting expression should be given a name. For instance, "nondet$(2,7) -$ nondet$(2,7)$" represents any value in $\{-5, 0, 5\}$, as both "nondet$(2,7)$" expressions may evaluate to different values; while "let $v =$ nondet$(2,7)$ in $v - v$" always evaluate to 0. In LAF (Section 3), we give a fresh name to every sub-expression, even if only the result of nondeterministic operations do require a name.

However, the necessity of giving new unique names to the result of joining expressions mean that this "symbolic expression abstraction" cannot have a lattice structure: each "join" of two terms generates a *different* least upper-bound of these two expressions.

We solve this issue simply by not requiring abstract domains to have a lattice structure. This choice allows us to handle join (and loops) precisely while existing lattice-based domains [13, 25, 4, 11, 22, 18] cannot. While this is very unusual, abstract interpretation is not necessarily lattice-based [8]: the core of abstract interpretation is to compute an abstract state which is sound with regards to the collecting semantics of the program, which is what our framework does (Section 4).

Widening does not fit symbolic abstract domains LAF introduces a μ operator that allows to *fold* expressions, which is necessary to have a finite abstraction in the presence of loops. For instance, the contents of `x` in the program `while(*) x = x + x;` is represented in LAF by "$(\mu x.x + x)(x_0)$", where x_0 represents the initial value of `x`. This expression represents a non-deterministic choice between "x_0", "$x_0 + x_0$", "let $x_1 = (x_0 + x_0)$ in $x_1 + x_1$", etc.

However, the standard operator used to find a finite abstraction in the presence of loops, widening, is not the right tool to find this folding. In essence, widening amounts to guessing a loop invariant given (a computation using) the first iterations of a loop. In the example above, the initial value of `x` is x_0, and the one at the beginning of the second iteration is $x_0 + x_0$. But there is an infinity of possible foldings that match these two iterations, including $(\mu x.x + x_0)(x_0)$; $(\mu x.x_0 + x_0 + 37 * (x - x_0))$; etc.; and this is even more difficult if expressions can be rewritten before widening!

The interface that we propose for fixpoint computation, based on the evaluation of LAF terms, takes as input the value at the loop entry (the argument of the μ expression), together with the *effect* of the loop (the body of the μ expression). This way, we always succeed in computing the folding of an expression, in a single step. Moreover, it is sufficiently general to also allow widening-like fixpoint computations; but also acceleration [15], policy iteration [6], inductive invariant generation [28]...

2.2 Benefits of abstract interpretation over symbolic expressions

LAF terms are purely functional expressions: the values to which they can evaluate is independent of a program point. Consequently, there is no need to split semantic information by program points as in classical analysis. Instead, semantic information can be put in a *single store*, which enables more sharing.

For instance, in Figure 1, the semantic store (a map from LAF variables to their values, according to conditions of the program) is represented at the right of the figure. The possible values for `nabs` after the join are centralized in the store, instead of being duplicated in the abstract element corresponding to the following program points.

One consequence of this single-store architecture is that transfer functions are more efficient. For instance the semantic store of Figure 1 can be implemented by an extensible array, and computing the interval of a variable is an amortized $\mathcal{O}(1)$ operation. Moreover, *nondet* can be seen as a join *targeting* the variables that have changed (`abs` and `nabs`). This optimization is essential, as most conditionals impact a small number of memory locations [3]; targeted join (and targeted fixpoint) alleviates the need for each abstract domain to detect the parts of the memory that have changed, improving the complexity of these operations (Section 4.2).

In the usual A.I., refining semantic information between statements is limited because the semantic information is split into program points. The single-store architecture removes this restriction. Section 5 details how, on Figure 1, the constraint propagation traverses the whole program to prove that $\mathtt{x} \in [-8; 8]$ at the end of the program (and the last assertion). The presence of *nondet* operation and its traversal are essential to relate `x` in the program to its absolute value `abs`. Moreover this constraint propagation is done using the LAF term, i.e. on the data dependencies of the program, automatically skipping statements unaffected by the refinement (like in sparse analyses [27]).

2.3 Hierarchy of translator domains

Our abstract domains are able to both abstractly interpret LAF terms, and produce a program abstraction as a LAF term. This allows to implement *translator domains*. Translator domains perform a *dynamic* translation of an input term into a simplified, possibly approximated, output term, using semantic information computed during the analysis (on either the input or output term).

One can then structure an abstract interpreter as a combination of (simple) translator domains. Superficially, this resembles the structure of compilers; however translator domains are not just sequential passes, but abstract domains executing simultaneously, and mutually refining each other.

Example: translator abstract domain for low-level memory operations Figure 2 shows how this can be applied to the static analysis of C programs with low-level memory manipulation such as pointer arithmetic, casts and bitfield (similarly to Miné [24]). The memory abstract domain represents memory regions as a contiguous sequence of slices. To each slice correspond the value that was last written. These values are represented as variables of a LAF term (in the theory of bitvectors). The right of the figure represents the memory region for `r` at different program points. Each write modifies the region by replacing the slice with the corresponding offset, size, and value.

The memory domain has to know the possible values for array indices. This is done by *querying* the underlying abstract domain for the possible values of

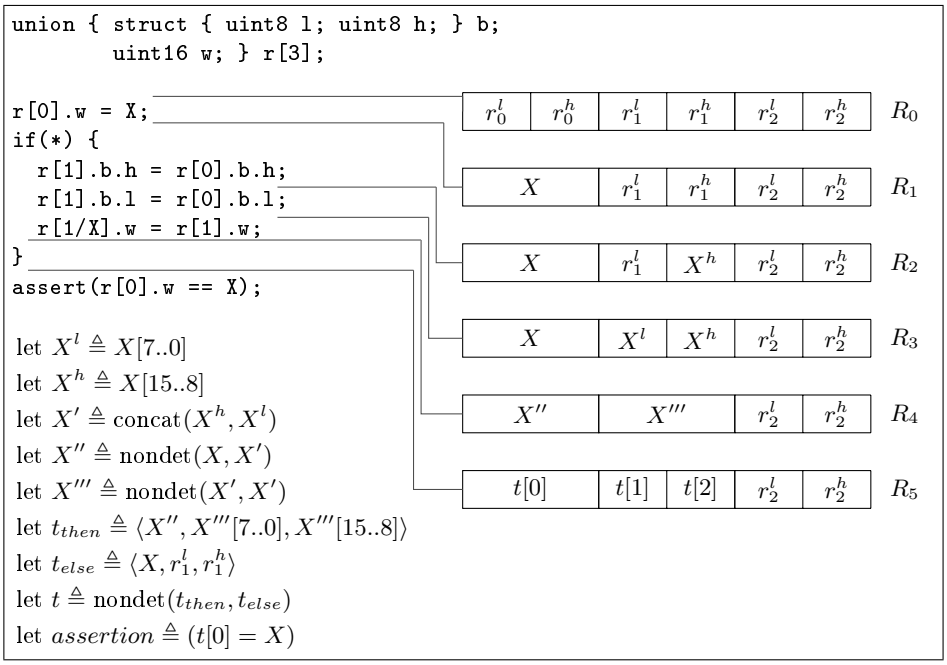

Fig. 2: A hierarchy of abstract domains handling low-level memory operations. The right part contains the abstract representation of the memory region for `r` at different program points. This representation contains variables of the LAF term (with bitvector theory) on the left. $\cdot[b..a]$ denotes substring extraction of a bitvector.

these indices. If there is more than one possible value (e.g. $1/X \in \{-1; 0; 1\}$), the abstract domain performs a weak update, i.e. the corresponding slice is mapped to a non-deterministic choice between the new and old value (R_4). On program joins, the slices are compared: only the cells that do not contain syntactically equal variables are "joined", which targets the join on memory cells that have been modified by the conditional.

Proving the assertion with symbolic reasoning Now, there are several ways to solve the final assertion. The first is to insert between these domains a syntactic rewrites abstract domains. Applying the following four simple rewriting rules suffices to prove the assertion (all primed versions of X are equal):

$$concat(x[a..b], x[b-1..c]) \;\rightarrow\; x[a..c] \qquad x[0..c] \text{ when } \text{sizeof}(x) = c \;\rightarrow\; x$$
$$\text{nondet}(x, x) \;\rightarrow\; x \qquad x == x \;\rightarrow\; true$$

Another mean is to output the corresponding formula to a specialized SMT solver, which is necessary for the most complex assertions. Contrary to the original program, the simplified LAF formula does not require memory operators: it only needs operator from the bitvector theory, that many SMT solver under-

stand. Thus, the memory abstract domain can be viewed as a translator from the source program to a simpler, approximate program, directly suitable for export to specialized solvers.

3 LAF: syntax and collecting semantics

The Logic of Approximation and Fixpoint (LAF) is the language we designed to be used both as a symbolic abstraction in abstract domains, and as an input for abstract domains. LAF can be viewed as a non-deterministic, functional language representing the possible results of a computation.

C is a context ($\in Context$), x a variable ($\in Var$), $\mathcal{S}$ a sort, t a term

$$
\begin{aligned}
C ::= {} & [] \\
| \;\; & \text{let } x \triangleq op_n(x,\ldots,x) \text{ in } C \\
| \;\; & \text{let } x \triangleq \text{nondet}(x,x) \text{ in } C \\
| \;\; & \text{let } x \triangleq (\mu x_s.\ C[x_e])(x_i) \text{ in } C \\
| \;\; & \text{let } x \triangleq \text{assume}(x_c,x_v) \text{ in } C \\
| \;\; & \text{let } x \triangleq \text{unknown}_{\mathcal{S}}() \text{ in } C \\
t ::= {} & C[x]
\end{aligned}
$$

Notation for theories:
Booleans ($\mathbb{B}$)**:** $\wedge, \vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$; true,false: $\mathbb{B}$; $\neg : \mathbb{B} \to \mathbb{B}$
Integers ($\mathbb{Z}$)**:** 0,1,...: $\mathbb{Z}$; $+, *, -, / : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$; $<, \leq : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$;
Tuples: $\pi_i : \langle S_1,\ldots,S_i,\ldots,S_n\rangle \to S_i$; $\langle\ldots\rangle : S_1 \times \ldots \times S_n \to \langle S_1,\ldots,S_n\rangle$;

Γ is an environment ($\in Env$). $\Gamma[x \mapsto v]$ and $\Gamma[x]$ respectively denote environment extension and lookup.

$$
\begin{aligned}
[\![[]]\!](\Gamma) &= \{\Gamma\} \\
[\![\text{let } x \triangleq op_n(x_1,\ldots,x_n) \text{ in } C]\!](\Gamma) &= [\![C]\!](\Gamma[x \mapsto op(\Gamma[x_1],\ldots,\Gamma[x_n])]) \\
[\![\text{let } x \triangleq \text{nondet}(x_1,x_2) \text{ in } C]\!](\Gamma) &= [\![C]\!](\Gamma[x \mapsto \Gamma[x_1]]) \;\cup\; [\![C]\!](\Gamma[x \mapsto \Gamma[x_2]]) \\
[\![\text{let } x \triangleq \text{unknown}_{\mathcal{S}}() \text{ in } C]\!](\Gamma) &= \bigcup_{v \in \mathcal{S}} [\![C]\!](\Gamma[x \mapsto v]) \\
[\![\text{let } x \triangleq \text{assume}(x_c,x_v) \text{ in } C]\!](\Gamma) &= \begin{cases} [\![C]\!](\Gamma[x \mapsto \bot]) & \text{if } \Gamma[x_c] = \text{false} \\ [\![C]\!](\Gamma[x \mapsto \Gamma[x_v]]) & \text{if } \Gamma[x_c] = \text{true} \end{cases} \\
[\![\text{let } x \triangleq (\mu x.\ C_b[x'])(x_0) \text{ in } C]\!](\Gamma) &= \text{let } \texttt{fix}(S) = \\
& \quad \text{let } S_b = \bigcup_{v \in S} [\![C_b]\!](\Gamma[x \mapsto v]) \text{ in} \\
& \quad \text{let } S' = \{\Gamma_b[x'] : \Gamma_b \in S_b\} \text{ in} \\
& \quad S_e \cup \{\Gamma[x_0]\} \\
& \text{in} \\
& \text{let } S = \text{lfp}(\texttt{fix}) \text{ in} \\
& \bigcup_{v \in S} \; [\![C]\!](\Gamma[x \mapsto v])
\end{aligned}
$$

Fig. 3: Syntax (top) and collecting semantics (bottom) of LAF

Syntax A LAF term (Figure 3) is essentially a sequence of *variable definitions*, followed by a single variable (the *result*). A variable is defined as the result of calling a primitive operation over previously-defined variables: i.e. every intermediary computation is *named*, and the sequence of definitions is *ordered*.

The terms need to be built incrementally; i.e. we want to extend a term with new definitions. We represent this formally using evaluation contexts C [31], i.e. a "term with a hole". The hole [] can be substituted with a variable x (to form a complete term) or with a context C_2 (to form a new context). For instance, if $C =$ "let $x \triangleq 12$; let $y \triangleq x + 1$; []" then $C[y]$ is a term evaluating to 13; C["let $z \triangleq x + y$ in []"] is a context appended with a new definition (to improve readability, we usually write the latter "C[let $z \triangleq x + y$]").

The primitive operations are those of a logical theory, (e.g. integer, floating point or real arithmetic; array; bitvectors; uninterpreted functions): there are many LAF languages, that depend on the theories in use. Two theories are always present: the theory of Boolean operations, and the theory of tuples.

Collecting Semantics Because variable definitions are ordered, we can define an operational semantics of LAF terms. LAF can be thus seen as a nondeterministic, functional language. A semantics of LAF given as a small-step structural operational semantics [29] exists (see Appendix A). But giving the collecting semantics for the term directly is actually simpler (see Figure 3); this illustrates the fact that LAF is a logic adequate for a symbolic description of a collecting semantics.

The semantics is defined using a *collecting evaluation* function $[\![\cdot]\!] : Context \times Env \to \mathcal{P}(Env)$. It takes a context $C \in Context$ remaining to be evaluated, an *environment* $\Gamma \in Env$ corresponding to a possible evaluation so far (more precisely, Γ should contain a mapping for all the free variables in C); and returns all the possible corresponding environments when C is fully evaluated. To evaluate a closed context (with no free variables), we pass the empty environment ε.

The collecting semantics essentially computes the set of all possible environments using meet-over-all-paths [26]. For non-deterministic constructs (*nondet*, *unknown*, and μ), every possible choice is fully evaluated in isolation, before taking the union of the possible outcomes. μ defines a local fixpoint of all the values obtained by an arbitrary number of iterations of the loop body $C_b[x']$, before making a nondeterministic choice of one of these values. In other words, μ is as an operator executing the loop body a nondeterministic number of times.

Intuitively, "assume(false,x)" is used to "kill" a part of the evaluation. A particularity of LAF is that the same evaluation can have some parts killed and some live (e.g. only one of the branch of a conditional is live). For instance, the desired result of evaluating the term "let $x \triangleq \text{assume}(false, 1)$; let $y \triangleq 2; y$" is 2. This is achieved using a special $\perp$ value to represent "killed" values. $\perp$ propagates throughout theory operations: e.g. $\perp + 3 = \perp$.

4 Abstract evaluation

4.1 Definition

We name *abstract evaluation* the abstract interpretation of functional LAF terms. An abstract domain is mainly composed of an *abstract evaluation function* $[\![\cdot]\!]^\sharp$ that evaluates contexts into *abstract states*; and a *concretization* function γ that gives a meaning to an abstract state as a set of environments. The main soundness rule is that the abstract state is always a superset of the set of possible environments of the context, as defined by the collecting semantics.

To allow the incremental building of the abstract value, the abstract evaluation function takes as an argument the abstract value computed so far, which overapproximates the set of environments containing the free variables of the context that remains to be evaluated. More formally:

Definition 1 (Abstract Domain). *An abstract domain is given by a quadruple* $\langle Env^\sharp, [\![\cdot]\!]^\sharp, \varepsilon^\sharp, \gamma : Env^\sharp \to \mathcal{P}(Env)\rangle$ *where:*
- $Env^\sharp$ *is a set, the set of* abstract states *(also called* abstract environments*);*
- $[\![\cdot]\!]^\sharp \in Context \times Env^\sharp \to Env^\sharp$ *is the* abstract evaluation function*;*
- $\varepsilon^\sharp \in Env^\sharp$ *is the* initial abstract state*;*
- $\gamma \in Env^\sharp \to \mathcal{P}(Env)$ *is the* concretization function.

Definition 2 (Soundness of Abstract Domains). *An abstract domain* $\langle Env^\sharp, [\![\cdot]\!]^\sharp, \varepsilon^\sharp, \gamma : Env^\sharp \to \mathcal{P}(Env)\rangle$ *is* sound *if the following rules apply:*

1. *Soundness of the initial state:* $\varepsilon \in \gamma(\varepsilon^\sharp)$
2. *Soundness of abstract evaluation:* $\forall C, \Gamma^\sharp : \bigcup_{\Gamma \in \gamma(\Gamma^\sharp)} [\![C]\!](\Gamma) \subseteq \gamma([\![C]\!]^\sharp(\Gamma^\sharp))$

Note how these definitions have replaced the traditional lattice-based structure of abstract domains by the sole evaluation of operators in the logic. Section 4.2 shows that traditional abstract domains fit into this definition; while Section 4.3 shows how the definition allows to also incorporate techniques not traditionally seen as abstract domains, for instance term rewriting.

4.2 Numerical abstract interpretation

A non-relational abstract domain is a mapping from variables to an abstraction of the value of this variable, that we call *abstract value*. Example of abstract values include intervals, congruences [16], the flat lattice of constants, the powerset of $\{\text{true}, \text{false}\}\ldots$ They have a lattice-based structure and can be combined using the operators of the LAF theories (e.g. $\dot{+}$ denotes the addition of two intervals).

Definition 3 (Abstract value). *An* abstract value $\mathcal{V}$ *is a pair* $\langle L_\mathcal{V}, \gamma_\mathcal{V}\rangle$ *where:*

1. $L_\mathcal{V}$ *is a lattice, equipped with join* ($\dot{\sqcup}$)*, inclusion* ($\dot{\sqsubseteq}$)*, widening* ($\dot{\nabla}$) *operations, as well as abstractions of theory operations* $\dot{op}$

2. $\gamma_\mathcal{V} : L_\mathcal{V} \to \mathcal{P}(Values)$ *concretizes elements of the lattice into a set of values;*

3. For every n-ary operation $\dot{op}$ *over lattice elements:*
$\gamma_\mathcal{V}(\dot{op}(L_1, \ldots, L_n)) \supseteq \{op(x_1, \ldots, x_n) : x_1 \in \gamma_\mathcal{V}(L_1), \ldots, x_n \in \gamma_\mathcal{V}(L_n)\}$

4. $\gamma_\mathcal{V}(L_1 \dot{\sqcup} L_2) \supseteq \gamma_\mathcal{V}(L_1) \cup \gamma_\mathcal{V}(L_2)$

Figure 4 presents a non-relational abstract domain equipped with such an abstract value, i.e. a concretization and an abstract evaluation algorithm which computes the over-approximation of the possible values of a LAF term.

$$
\begin{array}{ll}
\varepsilon^\sharp = & \varepsilon \\
[\![[]]\!]^\sharp(\Gamma^\sharp) = & \Gamma^\sharp \\
[\![\text{let } x \triangleq op_n(x_1, \ldots, x_n) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto \dot{op}(\Gamma^\sharp[x_1], \ldots, \Gamma^\sharp[x_n])]) \\
[\![\text{let } x \triangleq \text{nondet}(x_1, x_2) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto \Gamma^\sharp[x_1] \dot{\sqcup} \Gamma^\sharp[x_2]]) \\
[\![\text{let } x \triangleq \text{unknown}() \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto \dot{\top}]) \\
[\![\text{let } x \triangleq \text{assume}(x_c, x_v) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto \Gamma^\sharp[x_v]]) \\
[\![\text{let } x \triangleq (\mu x_s.\ C_b[x_e])(x_i) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & \text{let } L_i = \Gamma^\sharp[x_i] \text{ in} \\
& \text{let rec } \texttt{fix}(L) = \\
& \quad \text{let } \Gamma_s^\sharp = \Gamma^\sharp[x_s \mapsto L] \text{ in} \\
& \quad \text{let } \Gamma_b^\sharp = [\![C_b]\!]^\sharp(\Gamma_s^\sharp) \text{ in} \\
& \quad \text{let } L' = \Gamma_b^\sharp[x_e] \dot{\sqcup} L_i \text{ in} \\
& \quad \text{if } (L' \dot{\sqsubseteq} L) \text{ then } L \text{ else } \texttt{fix}(L \dot{\nabla} L') \\
& \text{in} \\
& [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto \texttt{fix}(L_i)])
\end{array}
$$

$$\Gamma \in \gamma(\Gamma^\sharp) \quad \Leftrightarrow \quad \forall (x \mapsto v) \in \Gamma :\ v \in \gamma_{\mathcal{V}}(\Gamma^\sharp[x])$$

$$Env^\sharp = Var \to L_{\mathcal{V}}$$

Fig. 4: A non-relational abstract domain, parametrized by an abstract value $\mathcal{V}$

Theorem 1. *Given that $\mathcal{V}$ is an abstract value, the quadruple $\langle Env^\sharp, [\![\cdot]\!]^\sharp, \varepsilon^\sharp, \gamma \rangle$ of Figure 4 is a sound abstract domain.*

The algorithm is quite straightforward: the abstract value is a standard mapping from program variables to an abstract value representing their possible values. Every evaluation step applies a lattice operation $\dot{op}$, except *nondet* which requires a join, and μ for which we do a local fixpoint computation. *assume* is ignored (for more precision, we could map x to $\bot$ when we detect that the condition cannot be true); Section 5 explains how the domain can be extended to handle *assume*.

The complexity of this implementation is *optimal*. Indeed, the $\Gamma^\sharp$ environment can be implemented using a single array (using variables as the indices), which means that environment update and lookup have $\mathcal{O}(1)$ complexity. If we assume that the abstract value of tuple values is represented as a tuple of scalar abstract values, and operations on scalar abstract values (e.g. interval) have $\mathcal{O}(1)$ complexity, then all operations have $\mathcal{O}(1)$ complexity, except joining and widening tuples of length n, which have $\mathcal{O}(n)$ complexity. Now, the length of the tuple depends on how it was generated; but if it was generated so that it contains only the variables that differ, then the complexity of these operations is $\mathcal{O}(\Delta)$, where Δ is the number of variables modified in a loop or in a conditional.

This improves on the complexity of analyzers using the traditional interface, for which the complexity of these operations depends (logarithmically) on the number of variables [2].

Theorem 2. *It is possible to lift traditional, lattice-based relational abstract domains to sound abstract domains following Definition 1.*

With minor modifications, the non-relational abstract domain of the previous section can be used to lift traditional abstract domains to our framework: it suffices to map memory variables to the usual lattices representing the entire memory. For instance, we can evaluate the definition
"let $M' = \text{store}(M, \texttt{"x"}, \text{load}(M, \texttt{"x"}) + 1)$" using the affine equality abstract domain [19], associating elements of this domain to M and M'.

It is also possible to use traditional abstract domains to relate variables of a (numeric) LAF term; Appendix B details how this lifting is done. It follows the same pattern as the non-relational lift: evaluation of "nondet" corresponds to join, and evaluation of μ corresponds to inclusion testing and widening.

4.3 Rewriting-based abstract interpretation

This section provides a simple example of an abstract domain whose abstract state is based on a LAF context, instead of a lattice.

The abstract domain is based on term rewriting. It performs a dynamic translation from a source term to a destination term, used as the abstract state; the concretization of this state is simply its collecting semantics. The domain is parametrized by a set of term rewriting rules $\mathcal{R}$.

$$
\begin{array}{ll}
\varepsilon^\sharp = & [] \\
[\![[]]\!]^\sharp(C^\sharp) = & C^\sharp \\
[\![\text{let } x \triangleq op(x_1, \ldots, x_n) \text{ in } C]\!]^\sharp(C^\sharp) = & [\![C]\!]^\sharp(C^\sharp[\text{let } x \triangleq \mathcal{R}(op(x_1, \ldots, x_n))]) \\
[\![\text{let } x \triangleq \text{nondet}(x_1, x_2) \text{ in } C]\!]^\sharp(C^\sharp) = & [\![C]\!]^\sharp(C^\sharp[\text{let } x \triangleq \mathcal{R}(\text{nondet}(x_1, x_2))]) \\
[\![\text{let } x \triangleq \text{unknown}_{\mathcal{S}}() \text{ in } C]\!]^\sharp(C^\sharp) = & [\![C]\!]^\sharp(C^\sharp[\text{let } x \triangleq \mathcal{R}(\text{unknown}_{\mathcal{S}}())]) \\
[\![\text{let } x \triangleq \text{assume}(x_c, x_v) \text{ in } C]\!]^\sharp(C^\sharp) = & [\![C]\!]^\sharp(C^\sharp[\text{let } x \triangleq \mathcal{R}(\text{assume}(x_c, x_v))]) \\
[\![\text{let } x \triangleq (\mu x_s.\ C_b[x_e])(x_i) \text{ in } C]\!]^\sharp(C^\sharp) = & \text{let } C_b^\sharp = [\![C_b]\!]^\sharp([]) \text{ in} \\
 & [\![C]\!]^\sharp(C^\sharp[\text{let } x \triangleq \mathcal{R}((\mu x_s.\ C_b^\sharp[x_e])(x_i))])
\end{array}
$$

$$\Gamma \in \gamma(C^\sharp) \quad \Leftrightarrow \quad \Gamma \in [\![C^\sharp]\!](\varepsilon)$$

Fig. 5: Rewriting abstract domain parametrized by a term rewriting system $\mathcal{R}$

We give term rewriting rules using pattern matching, on a representation of terms where let bindings have been inlined. Simple term rewriting rules include $x \wedge x \to x$, $1 * x \to x$, $2 + x + 3 \to x + 5$. However, as is, rules such as $0 * x \to 0$ or $x - x \to 0$ are not correct for this domain. Indeed, consider the terms:

"let $u \triangleq \text{unknown}_{\mathbb{Z}}$ in let $v \triangleq \text{assume}(u \neq 3, u)$ in let $s \triangleq 0 * v$ in s".

and the result of applying the rule $0 * x \to 0$ to it:

"let $u \triangleq \text{unknown}_{\mathbb{Z}}$ in let $v \triangleq \text{assume}(u \neq 3, u)$ in let $s \triangleq 0$ in s".

The possible environment $[u \mapsto 3, v \mapsto \bot, s \mapsto \bot]$ for the first term has been replaced by the environment $[u \mapsto 3, v \mapsto \bot, s \mapsto 0]$ in the second. One way to deal with this issue is to propagate the "assume" conditions in the replacement. Another is to accept these "over-approximating rewrites", by changing the definition of γ to:

$$\Gamma \in \gamma(C^\sharp) \quad \Leftrightarrow \quad \exists \Gamma' \in [\![C^\sharp]\!](\varepsilon) : \forall x \in \Gamma : \Gamma[x] = \Gamma'[x] \vee \Gamma[x] = \bot \qquad (1)$$

The latter allows very aggressive rewrites, such as $x/x \to 1$ or $1/x < 2 \to \text{true}$, that disregard side conditions, usually a major difficulty of term-rewriting.

The soundness of both "exact" and "over-approximating" abstract domains is established by considering only the term rewriting rules:

Definition 4. *A term rewriting rule $l \to r$ is an* exact *rewrite if, for any substitution σ of the free variables of l, the evaluation of $l\sigma$ equals the evaluation of $r\sigma$. It is* over-approximating *if they are equal when the evaluation of $l\sigma$ does not return $\bot$.*

Theorem 3. *If every term rewriting rule in $\mathcal{R}$ is* exact*, then the abstract domain of Figure 5 is sound. If every term rewriting rule in $\mathcal{R}$ is* over-approximating*, then the abstract domain of Figure 5, with γ given by equation 1, is sound.*

The term rewriting and non-relational domain represent very different domains, one semantic in nature, the other more symbolic. We will now see a more complex domains that combine and extend theses two models: the constraint propagation abstract domain.

5 A constraint propagation abstract domain

The constraint domain is a good example of the advantages of our approach: benefiting from the structure of LAF terms (including targeted join and widening) that allows a single store implementation, it can propagate and combine semantic information across the whole program, in an efficient way. It is made of three elements:

- A LAF context $C^\sharp$, called the *constraints* (middle of Figure 1). It can be seen as a rewrite of the input LAF context with particular locations for *assume* definitions.
- Two mappings $\Gamma_c^\sharp$ and $\Gamma_v^\sharp$ from variables of the input context to variables of the constraints. This mapping is made such that x and "$\text{assume}(\Gamma_c^\sharp(x), \Gamma_v^\sharp(x))$" evaluate to the same values (the mapping and input term are not shown on Figure 1).4
- A mapping M from each variable of the constraints to a *condition map*, representing the possible values for the variable according to some conditions (right of Figure 1). How these conditions are chosen depends on a strategy; ours is detailed below.

Generation of constraints The construction of the constraints, $\Gamma_c^\sharp$ and $\Gamma_v^\sharp$ is pretty straightforward (Figure 6). The general idea is that the constraints could represent the input term stripped from assume expressions; the condition is instead stored in the $\Gamma_c^\sharp$ map. Actually we do not strip the assume statements entirely, but delay them until the end of loops, or just before a "nondet", so that the term used as the constraints does not lose information with regards to the input term.

$$
\begin{array}{ll}
\varepsilon^\sharp = & [] \qquad (x' \text{ is fresh}) \\
[\![[]]\!]^\sharp(C^\sharp, \Gamma_v^\sharp, \Gamma_c^\sharp) = & \langle C^\sharp, \Gamma_v^\sharp, \Gamma_c^\sharp \rangle \\
[\![\text{let } x \triangleq op(x_1, \ldots, x_n) \text{ in } C]\!]^\sharp(C^\sharp, \Gamma_v^\sharp, \Gamma_c^\sharp) = & [\![C]\!]^\sharp(\, C^\sharp[\text{let } x' \triangleq op(\Gamma_v[x_1], \ldots, \Gamma_v[x_n])], \\
 & \qquad \Gamma_v^\sharp[x \mapsto x'], \Gamma_c^\sharp[x \mapsto \Gamma_c^\sharp[x_1] \wedge \ldots \wedge \Gamma_c^\sharp[x_n]]) \\
[\![\text{let } x \triangleq \text{nondet}(x_1, x_2) \text{ in } C]\!]^\sharp(C^\sharp, \Gamma_v^\sharp, \Gamma_c^\sharp) = & [\![C]\!]^\sharp(\, C^\sharp[\text{let } x' \triangleq \text{nondet}(\, \text{assume}(\Gamma_c^\sharp[x_1], \Gamma_v^\sharp[x_1]), \\
 & \qquad\qquad \text{assume}(\Gamma_c^\sharp[x_2], \Gamma_v^\sharp[x_2]))], \\
 & \qquad \Gamma_v^\sharp[x \mapsto x'], \Gamma_c^\sharp[x \mapsto \Gamma_c^\sharp[x_1] \vee \Gamma_c^\sharp[x_2]]) \\
[\![\text{let } x \triangleq \text{unknown}_{\mathcal{S}}() \text{ in } C]\!]^\sharp(C^\sharp, \Gamma_v^\sharp, \Gamma_c^\sharp) = & [\![C]\!]^\sharp(\, C^\sharp[\text{let } x' \triangleq \text{unknown}_{\mathcal{S}}()], \\
 & \qquad \Gamma_v^\sharp[x \mapsto x'], \Gamma_c^\sharp[x \mapsto \text{true}]) \\
[\![\text{let } x \triangleq \text{assume}(x_c, x_v) \text{ in } C]\!]^\sharp(C^\sharp, \Gamma_v^\sharp, \Gamma_c^\sharp) = & [\![C]\!]^\sharp(\, C^\sharp, \Gamma_v^\sharp[x \mapsto \Gamma_v^\sharp[x_v]], \\
 & \qquad \Gamma_c^\sharp[x \mapsto \Gamma_c[x_c] \wedge \Gamma_c[x_v] \wedge \Gamma_v[x_c]]) \\
[\![\text{let } x \triangleq (\mu x_s.\ C_b[x_e])(x_i) \text{ in } C]\!]^\sharp(C^\sharp, \Gamma_v^\sharp, \Gamma_c^\sharp) = & \text{let } \langle C_b^\sharp, \Gamma_v'^\sharp, \Gamma_c'^\sharp \rangle = [\![C_b]\!]^\sharp([], \Gamma_v^\sharp, \Gamma_c^\sharp) \text{ in} \\
 & \text{let } E = \text{"assume}(\Gamma_c'^\sharp[x_e], \Gamma_v'^\sharp[x_e])\text{" in} \\
 & [\![C]\!]^\sharp(\, C^\sharp[\text{let } x' \triangleq (\mu x_s.\ C_b^\sharp[E])(\Gamma_v^\sharp[x_i])], \\
 & \qquad \Gamma_v^\sharp[x \mapsto x'], \Gamma_c^\sharp[x \mapsto \Gamma_c^\sharp[x_i]])
\end{array}
$$

Fig. 6: Generation of the constraints $C^\sharp$ and the maps $\Gamma_c^\sharp$ and $\Gamma_v^\sharp$.

Initial evaluation We now describe the construction of the mapping M of condition maps. An *initial evaluation* of an input variable x is done concurrently with constraints generation: the possible values for $\Gamma_v^\sharp[x]$ is computed using its definition, for the condition $\Gamma_c^\sharp[x]$. For instance in Figure 1, the initial evaluation of $xdiv$ is done with condition c_2; we first retrieve the abstract value for x with condition c_2 (which is $[-8; -1] \dot{\sqcup} [0; 8] = [-8; 8]$); then the value for 9 (which is $[9; 9]$); then create the binding $xdiv \mapsto c_2 \Vdash [0; 0]$, where $[0; 0] = [-8; 8] \dot{/} [9; 9]$.

Constraint propagation When an assume(c, x) definition is evaluated in the input term, we perform a *constraint propagation*. It consists in reevaluating definitions such as "let $c \triangleq x < 0$", refining and using the information attached to the variables corresponding to the result and arguments of the operator. The algorithm is similar to constraint satisfaction algorithms such as AC-3 [23]; but we maintain, together with the worklist of variables whose abstract value has changed, the condition for which they have changed. The constraint propagation of condition c is initiated by adding the binding $c \mapsto c \Vdash \{true\}$. In Figure 1, a first chain of constraints propagation is the addition of the bindings $c_1 \mapsto c_1 \Vdash \{true\}$, then $x \mapsto c_1 \Vdash [-\infty; -1]$.

When an assume$(c', \ldots)$ constraint is traversed, the c' condition is added as a conjunct to the condition being propagated. This is seen in the addition of

c_1 in the constraint propagation chain $c_2 \mapsto c_2 \Vdash \{true\}$, $abs \mapsto c_2 \Vdash [0;8]$, $nx \mapsto c_1 \wedge c_2 \Vdash [1;8]$, $x \mapsto c_1 \wedge c_2 \Vdash [-8;-1]$.

The constraint propagation phase terminates if the lattices used in the abstract value cannot be indefinitely refined (e.g. refining $y \leq y/2$ using an interval of rational numbers). But it is always sound to limit the number of propagation, and we evaluate different heuristics in Section 7.

Loops are handled like in the non-relational analysis of Section 4.2; the condition map, seen as a function lattice from conditions to abstract values, is used to join, widen and test for inclusion the loop input and output. The only difference is that the conditions defined inside the loop body do not have any meaning in the next iteration, or outside of the loop; thus as soon as the loop body has been fully evaluated, these conditions are removed by existential quantification.

Concretization The concretization is best defined as the composition of two parts. The first relates the input term to the generated constraints, and is similar to the one of rewriting-based abstract domains of Section 4.3.

$$\begin{array}{l} \Gamma \in \gamma_1(\langle C^\sharp, \Gamma_c^\sharp, \Gamma_v^\sharp \rangle) \;\Leftrightarrow \\ \exists \Gamma' \in [\![C^\sharp]\!](\varepsilon) : \forall x \mapsto v \in \Gamma : \begin{cases} \Gamma'[\Gamma_c^\sharp[x]] = \text{true} \;\wedge\; \Gamma'[\Gamma_v^\sharp[x]] = v & \text{if } v \neq \bot \\ \Gamma'[\Gamma_c^\sharp[x]] = \text{false} & \text{if } v = \bot \end{cases} \end{array}$$

The second part relates the term in the constraints to the values contained in the map M. This amounts to seeing constraint generation as a mere preprocessing of the input. It is similar to that of the non-relational abstract domain of 4.2, but taking conditions into account.

$$\Gamma \in \gamma_2(M) \quad \Leftrightarrow \quad \forall x \in \Gamma, c \Vdash v^\sharp \in M[x] : \; \Gamma[c] = \text{true} \;\Rightarrow\; \Gamma[x] \in \gamma_{\mathcal{V}}(v^\sharp)$$

The combination consists in replacing, in the definition of γ_1, the set of possible environments $[\![C^\sharp]\!](\varepsilon)$, by the approximation of this set $\gamma_2(M)$.

Theorem 4. *The constraint propagation abstract domain is sound.*

6 An abstract interpreter of embedded C programs

This section demonstrates the practical applicability of our approach, by describing the implementation of a complete analyzer for embedded C programs (including low-level memory manipulation such as casts and bitfields, but currently excluding recursion and dynamic memory allocation). The system is composed as a succession of simple abstract domain "passes" communicating with one another.

Figure 7 presents a high-level view of the analyzer. Each rectangular node is an abstract domain, that inputs a LAF term, and possibly outputs a simplified one. Input and output terms may use operators of different theories. Especially, the LAF term obtained from the translation of the C program contains `load` and `store` memory operators; while the LAF term used as input of the leaves abstract domains do not. This allows 1. to translate this term to SMT solvers

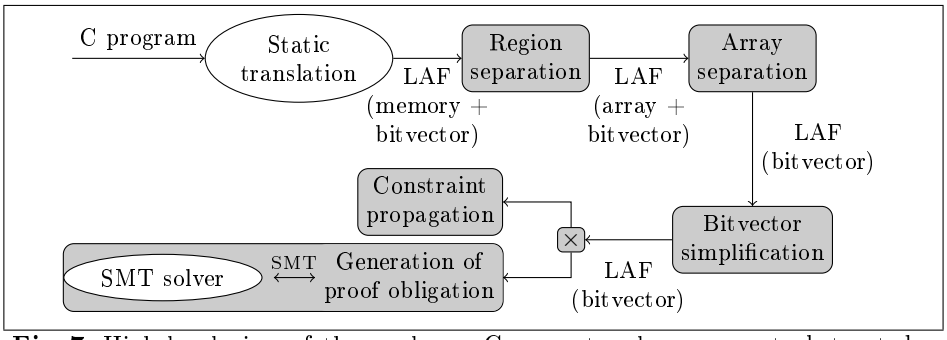

Fig. 7: High-level view of the analyzer. Gray rectangles represent abstract domains, and ellipses other processes. $\times$ represents abstract domain product: both domains have the same input term.

that only understand bitvector theory, and 2. to implement a numeric constraint propagation domain that is unaware of memory operations.

The translation from C to LAF is standard (similar to e.g. Cytron et al. [10]). Appendix D presents the full translation rules for the simpler WHILE language.

Region and cell separation The region separation functor splits the memory into independent, non-overlapping memory regions, where a region corresponds to the memory allocated for a C variable (local or global). The cell separation functor partitions a memory region into contiguous slices of known size, and was informally described in Section 2.3. Both are currently quite naive; but better memory representations functors (e.g. [5, 9]) could be adapted to become translator domains.

The output of these successive translator domains is a LAF term that does not contain memory operations, and whose variables correspond to the values contained in memory. This term is analyzed numerically by the later domains; the result of this analysis is itself used by the memory translator domains to know which memory locations are being read and written.

Purely syntactic term rewriting Our prime use of term rewriting is to simplify the bitvector concatenation and extraction operations. This happens notably when the program performs byte per byte copies (e.g. using `memcpy`). It plays the same role as the memory equality predicate domain of Miné [24], but the implementation is very different, as equality between values is provided by syntactic equality between LAF variables.

Constraint propagation was already described in section 5

Generation of proof obligations as first-order and Horn formulas translate LAF terms into a set of clauses, such that satisfiability testing allows to test if a variable can have a specific value. Appendix C details the formal translation of LAF term into 1st-order and Horn clauses; the translation to Horn clauses is exact, while the translation to first-order clauses is an over-approximation.

Note that this results in a true combination of abstract domains, and not just a first static analysis pass followed by generation of proof obligations. The

benefit is that an abstract domain can be combined with others so that the combination is mutually beneficial. In particular:

1. The generation of proof obligation domain benefits from the simplification of translator domains. In the analyzer, the generated SMT formula only refers to bitvector operations, and never to memory operations; this is beneficial since SMT solvers are much better at handling integers and bitvectors than at handling memory operations.
2. The domain also indirectly benefits from the numeric constraint propagation domain, as it is used by the translator domains to perform simplifications. It could also use the invariants computed by that domain to increase precision or speed up resolution.
3. In turn, the domain can be used to simplify complex boolean expressions into the *true* or *false* constants, allowing for a precision increase in the other domains. It can also be used e.g. to refine the bounds of intervals.

7 Experimental evaluation

This section presents preliminary benchmarks of our analyzer. Please keep in mind that it is still a prototype; it lacks some of the features of C (dynamic memory allocation, recursion, function pointers and `longjmp`); the handling of function calls (by recursive inlining) is still naive; no optimization of hot code path was performed.

Propagation limit				0		1			2			∞ (backward)		
Bench	#LOC	#expr	#A	time	UP	time	UP	improv.	time	UP	improv.	time	UP	improv.
adpcm	610	499	1249	1.12	39	0.94	39	56%	1.04	22	18%	1.06	22	0.0%
2048	435	617	8778	38.5	314	32.9	314	19%	33.19	100	94%	32.56	100	0.6%
papabench0.4	4983	8286	6415	84.4	412	78.2	412	0.9%	82.14	83	30%	80.96	83	0.0%
transport	13294	10674	11064	213	100	182	100	8.3%	61.41	14	16%	63.04	14	0.1%
robotics #0	13353	161	306	1.95	18	1.74	18	26%	1.79	18	0.0%	1.80	18	3.7%
robotics #1	13353	1454	1978	4.06	80	3.61	80	1.9%	2.85	51	14%	2.77	51	0.7%
robotics #2	13353	22	125	0.88	0	0.78	0	0.0%	0.84	0	0.0%	0.85	0	0.0%
robotics #3	13353	872	860	0.97	0	0.85	0	1.1%	0.94	0	0.0%	0.94	0	0.0%
robotics #4	13353	298	620	3.24	106	2.76	106	8.1%	1.46	62	76%	1.37	21	22%
robotics #5	13353	2372	375	98.2	221	85.9	221	3.2%	77.68	167	27%	74.70	165	0.9%
robotics #6	13353	280	481	0.96	6	0.85	6	0.0%	0.94	6	0.0%	0.92	6	0.0%

#LOC = lines of code; #expr = live expressions;
#A = total alarms to be proved; UP = alarms left unproved;
improv. = % of expressions more precise than the preceding propagation limit.

Table 1. Benefits of constraint propagation.

Analyzing embedded industrial applications Table 1 demonstrates the benefits of using constraint propagation on a variety of real benchmarks. The first columns give the number of expressions of the program that are not dead,

and the number of memory-related alarms that must be proved (i.e. pointers are valid and indices are in bounds). Then, for different limits on the constraint propagation, we give the analysis time (in seconds); the number of alarms that remain unproved; and the number (and percentage) of expressions for which we compute a more precise set of possible values, wrt. the preceding propagation limit. The first three benchmarks are open-source (2048 is a game; adpcm a filter; papabench0.4 represents code embedded in typical UAVs). All the other benchmarks are automatic control systems coming from various industries (the last is made of 7 independent threads).

The benchmarks demonstrate that constraint propagation has a real impact on the precision of the analysis. Going from 0 to 2 variables being propagated leads to refining most of the expressions (and alarms) of the programs, while *decreasing* the analysis time. This can be explained by the fact that less dead code is visited, but also in our case that memory updates are more precise and concern less locations. Going for unlimited backward propagation (or backward + forward, not shown in the table) also keep on being faster and more precise, but to a lesser degree. Note that the number of alarms is still high, but can be reduced to almost 0 by standard tricks such as loop unrolling and user annotations.

Generation of Horn clauses Another way to discharge unproved assertions is to send the remaining ones to a solver which focuses on a single goal. We made an experiment with benchmarks of the SVComp 2016 competition [1], whose goal is to prove the validity of an assertion in a program which range from few to tens of KLOCS. Most of these assertions are out of the reach of our abstract domain using intervals. However, the structure of out interpreter allows to build a LAF term, which is equivalent to the original program, but stripped from any memory operations. This property allows to leverage (after conversion, see Appendix C) the Z3 horn checker μZ [17], which does not support array theory, only bitvectors or integers.

	Lines of code			Buggy programs			Correct programs			
SVComp category	min	avg	max	total	proved	unsound	total	proved	unsound	points
Product lines	838	1943	3789	265	92	0	332	262	0	616
Loops	14	46	1644	48	18	0	93	42	0	102
ControlFlow	94	634	2152	18	3	0	30	15	0	33

Table 2. Number of programs proved buggy or correct, in some SVComp categories, with a 10s timeout. We count 2 points for every correct program proved correct, and 1 for buggy program proved buggy. Our tool never provides a wrong (unsound) answer.

On the two first categories and with a 10s timeout, the abstract interpreter combined with μZ is already competitive[1] (would rank 3rd/18 on Loops, and 3rd/16 on ControlFlow). This is despite all the shortcomings of our current

[1] `https://sv-comp.sosy-lab.org/2016/results/results-verified/`

implementation: some C features are not supported (e.g. variable length arrays), and we do not yet export the invariants we found to help in the Horn clauses.

8 Related work

The closest relatives to our term-based abstract interpretation framework are existing symbolic abstract domains in the traditional lattice-based framework.

The symbolic constant domain of Miné [25] maps program variables to expressions on other program variables, and provides a solution to the loss of precision induced by storing intermediate computations in temporary variables. Logozzo and Fähndrich [22] and Djoudi et al. [11] implement similar domains, and insists on the need of these methods for low-level languages, in which every computation makes use of temporary variables.

Chang and Leino [4] introduces a symbolic abstract domain where variables represent values, instead of memory locations. This avoids loosing precision when program variables are overwritten. Numeric abstract domains are used to compute relations between these symbolic variables. Chang and Rival [5] shows the importance of having variables representing values when designing precise memory abstractions. In term-based abstract interpretation, base abstract domains compute relations between LAF variables (which represent values); these variables are referred to by our memory abstraction. The main difference is that all the LAF variables are linked together in a term which is an abstraction of the whole program, i.e. we never lose any symbolic information, even on loops and control-flow joins.

Gange et al. [13] combines their symbolic abstract domain with constraint propagation over non-relational domains. As LAF terms represent loops and conditionals precisely, our constraint propagation abstract domain extends this work with the ability to propagate the constraints across the whole program.

All these domains belong to the traditional lattice-based framework; thus none of them leverage the fact that variables represent values by sharing all the information about them in a single store.

9 Conclusion

We have presented a term-based abstract interpretation framework, whose main ingredients are: a logic, that can be used as the abstract state in an abstract domain, and can represent the relations between values in the program without loss of precision; and the definition of abstract domains as abstract interpreters over this logic, allowing the definition of abstract domains as a combination of translations. We have demonstrated the applicability of the framework by describing several abstract domains combining numeric and symbolic reasoning; and we used these domains to build a complete analyzer for C programs. We now plan on applying the technique on languages where symbolic reasoning is very important, such as Static Single Assignment or binary analysis.

Bibliography

[1] Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 887–904. Springer (2016)

[2] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones 2566, 85–108 (Oct 2002), `http://www.di.ens.fr/~mine/publi/BlanchetCousotEtAl-LNCS-v2566-p85-108-2002.pdf`

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03). pp. 196–207. ACM Press, San Diego, California, USA (June 7–14 2003)

[4] Chang, B.Y.E., Leino, K.R.M.: Abstract Interpretation with Alien Expressions and Heap Structures, pp. 147–163. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

[5] Chang, B.Y.E., Rival, X.: Modular Construction of Shape-Numeric Analyzers. In: Banerjee, A., Danvy, O., Doh, K.G., Hatcliff, J. (eds.) Festschrift for Dave Schmidt. Festschrift for Dave Schmidt, vol. 129. EPTCS, Manhattan, Kansas, United States (Sep 2013)

[6] Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Computer aided verification. pp. 462–475. Springer (2005)

[7] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 238–252. ACM Press, New York, NY, Los Angeles, California (1977)

[8] Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (auf 1992)

[9] Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Conference Record of the 38^{th} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 105–118. ACM Press, New York, Austin, Texas (jan " 26-28" 2011)

[10] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (oct 1991)

[11] Djoudi, A., Bardin, S., Goubault, É.: Recovering high-level conditions from binary programs. In: FM 2016: Formal Methods - 21st International Sym-

posium, Limassol, Cyprus, November 9-11, 2016, Proceedings. pp. 235–253 (2016)
[12] Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software. pp. 10–30. FoVeOOS'10, Springer-Verlag, Berlin, Heidelberg (2011)
[13] Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An Abstract Domain of Uninterpreted Functions, pp. 85–103. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
[14] Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Exploiting sparsity in difference-bound matrices. In: Proceedings of the 23rd Static Analysis Symposium (SAS 2016) (2016)
[15] Gonnord, L., Schrammel, P.: Abstract acceleration in linear relation analysis. Science of Computer Programming 93, 125–153 (2014), author version : http://hal.inria.fr/hal-00787212/en
[16] Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics 30(3-4), 165–190 (1989)
[17] Hoder, K., Bjørner, N.: Generalized Property Directed Reachability, pp. 157–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
[18] Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified c static analyzer. ACM SIGPLAN Notices 50(1), 247–259 (2015)
[19] Karr, M.: Affine relationships among variables of a program. Acta Informatica 6, 133–151 (1976)
[20] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: a software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (2015)
[21] Leino, K.: Efficient weakest preconditions. Information Processing Letters 93(6), 281–288 (2005)
[22] Logozzo, F., Fähndrich, M.: On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis, pp. 197–212. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
[23] Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence 8(1), 99 – 118 (1977)
[24] Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06). pp. 54–63. ACM (Jun 2006), `http://www.di.ens.fr/~mine/publi/article-mine-lctes06.pdf`
[25] Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. CoRR (2007)
[26] Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer-Verlag New York Incorporated (1999)
[27] Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for c-like languages. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 229–238. PLDI '12, ACM, New York, NY, USA (2012)

[28] de Oliveira, S., Bensalem, S., Prevosto, V.: Polynomial invariants by linear algebra. In: Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. pp. 479–494 (2016)
[29] Plotkin, G.: A structural approach to operational semantics. Tech. rep., Aarhus University (1981)
[30] Venet, A., Brat, G.P.: Precise and efficient static array bound checking for large embedded C programs. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004. pp. 231–242 (2004)
[31] Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and computation (1994)

A Small-step operational semantics of LAF

A small-step operational semantics also exists for LAF terms:

$$\Sigma :: \langle \Gamma, \text{let } x = op_n(x_1, \ldots, x_n) \text{ in } t\rangle \rightarrow \\ \Sigma :: \langle \Gamma[x \mapsto op_n(\Gamma[x_1], \ldots, \Gamma[x_n])], t\rangle$$

$$\Sigma :: \langle \Gamma, \text{let } x = \text{nondet}(x_1, x_2) \text{ in } t\rangle \rightarrow \\ \Sigma :: \langle \Gamma[x \mapsto \Gamma[x_1]], t\rangle \\ \Sigma :: \langle \Gamma, \text{let } x = \text{nondet}(x_1, x_2) \text{ in } t\rangle \rightarrow \\ \Sigma :: \langle \Gamma[x \mapsto \Gamma[x_2]], t\rangle$$

$$\Sigma :: \langle \Gamma, \text{let } x = \text{assume}(x_c, x_v) \text{ in } t\rangle \rightarrow \\ \Sigma :: \langle \Gamma[x \mapsto \Gamma[x_v]], t\rangle \text{ when } \Gamma[x_c] = true \\ \Sigma :: \langle \Gamma, \text{let } x = \text{assume}(x_c, x_v) \text{ in } t\rangle \rightarrow \\ \Sigma :: \langle \Gamma[x \mapsto \bot], t\rangle \text{ when } \Gamma[x_c] = false$$

$$\Sigma :: \langle \Gamma, \text{let } x = (\mu x_s.\ t_b[x_e])(x_i) \text{ in } t\rangle \rightarrow \qquad \text{(do not enter loop)} \\ \Sigma :: \langle \Gamma[x \mapsto \Gamma[x_i]], t\rangle \\ \Sigma :: \langle \Gamma, \text{let } x = (\mu x_s.\ t_b[x_e])(x_i) \text{ in } t\rangle \rightarrow \qquad \text{(enter loop)} \\ \Sigma :: \langle \Gamma, \text{let } x = (\mu x_s.\ t_b[x_e])(x_i) \text{ in } t\rangle :: \langle \Gamma[x_s \mapsto \Gamma[x_i]], t_b[x_e]\rangle \\ \Sigma :: \langle \Gamma, \text{let } x = (\mu x_s.\ t_b[x_e])(x_i) \text{ in } t\rangle :: \langle \Gamma', x_e\rangle \rightarrow \qquad \text{(loop exit)} \\ \Sigma :: \langle \Gamma[x \mapsto \Gamma'[x_e]], t\rangle \\ \Sigma :: \langle \Gamma, \text{let } x = (\mu x_s.\ t_b[x_e])(x_i) \text{ in } t\rangle :: \langle \Gamma', x_e\rangle \rightarrow \qquad \text{(loop again)} \\ \Sigma :: \langle \Gamma, \text{let } x = (\mu x_s.\ t_b[x_e])(x_i) \text{ in } t\rangle :: \langle \Gamma[x_s \mapsto \Gamma'[x_e]], t_b[x_e]\rangle$$

The semantics is nondeterministic because of the constructors *nondet* and μ. The semantics uses a stack, whose depth represents the level of loop nesting in which we are. The stack is used to save the context when entering a loop. Elements of the stack are pairs of an environment and a term, respectively representing the values already computed, and the term that remains to be computed, for each loop nesting level. Most step just update the head of the stack, by updating the environment and the term.

Execution "blocks" when an `assume` expression is encountered, with a value of *false* for its first argument. This is represented by adding to all sorts a special $\bot$ symbol. Note that `assume` delimits the part of the term which is blocked; execution of other subterms continue without any problem.

Theorem 5 (Alternative definition of the collecting semantics). *We note by $\rightarrow^*$ the transitive closure of $\rightarrow$. Then*

$$[\![C]\!](\varepsilon) = \{\ \Gamma : \langle \varepsilon, C\rangle \rightarrow^* \langle \Gamma, []\rangle\ \}$$

B Lifting traditional relational abstract domains

When generating constraints (Figure 6), for each input variable x we extracted a condition $\Gamma_c^\sharp[x]$, corresponding to the necessary condition for x to evaluate to a value different from $\bot$. The idea here is similar: we associate to each input variable, an element D of a traditional abstract domain; this element corresponds to environments that match the condition $\Gamma_c^\sharp[x]$, and describes the relations between x and all the variables on which it depends transitively (where a depends on b means that b is an argument of the operator used to define a).

We note by γ_t the concretization of the traditional domain. The concretization is defined as follows: for every binding $x \mapsto v$ of possible environments Γ, every abstract domain element in $\Gamma^\sharp$ must agree that this binding is indeed possible.

$$\Gamma \in \gamma(\Gamma^\sharp) \quad \Leftrightarrow \quad \forall (x \mapsto v) \in \Gamma : \forall (y \mapsto D) \in \Gamma^\sharp : \exists \Gamma' \in \gamma_t(D) : \Gamma'[x] = v$$

Theorem 6. *If the original abstract domain is sound with regards to its concretization γ_t, then its lifting is also sound.*

Because this lifting relates all the variables of a LAF term; LAF terms can contain a large number of variables; most operations on numerical abstract domains have a complexity supra-linear in the number of variables, a naive application of this technique would probably be very slow. However, this can be mitigated by exploiting the fact that most variables would be unrelated [14], or limiting relations by *packing* variables together [3, 30].

$$
\begin{array}{ll}
\varepsilon^\sharp = & \varepsilon \\
\\
[\![\text{let } x = op(x_1, \ldots, x_n) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & \text{let } D = \bigsqcap_{1 \leq i \leq n} \Gamma^\sharp[x_i] \text{ in} \\
& \text{let } D' = \{|x \leftarrow op(x_1, \ldots, x_n)|\}(D) \text{ in} \\
& [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto D']) \\
\\
[\![\text{let } x = \text{nondet}(x_1, x_2) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & \text{let } D_1 = \{|x \leftarrow x_1|\}(\Gamma^\sharp[x_1]) \text{ in} \\
& \text{let } D_2 = \{|x \leftarrow x_2|\}(\Gamma^\sharp[x_2]) \text{ in} \\
& \text{let } D' = D' = D_1 \sqcup D_2 \text{ in} \\
& [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto D']) \\
\\
[\![\text{let } x = \text{unknown}() \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto \top]) \\
\\
[\![\text{let } x = \text{assume}(x_c, x_v) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & \text{let } D = \Gamma^\sharp[x_c] \sqcap \Gamma^\sharp[x_v] \text{ in} \\
& \text{let } D' = \{|\text{assume } x_c|\}(D) \text{ in} \\
& \text{let } D'' = \{|x \leftarrow x_v|\}(D') \text{ in} \\
& [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto D'']) \\
\\
\texttt{killall}([], D) = & D \\
\texttt{killall}(\text{let } x = ... \text{ in } C, D) = & \texttt{killall}(C, \{|x \leftarrow \text{unknown}()|\}(D)) \\
\\
[\![\text{let } x = (\mu x_s.\ C_b[x_e])(x_i) \text{ in } C]\!]^\sharp(\Gamma^\sharp) = & \text{let } D_i = \{|x \leftarrow x_i|\}(\Gamma^\sharp[x_i]) \text{ in} \\
& \text{let rec } \texttt{fix}(D) = \\
& \quad \text{let } D_s = \{x_s \leftarrow x\}(D) \text{ in} \\
& \quad \text{let } \Gamma_s^\sharp = \Gamma^\sharp[x_s \mapsto D_s] \text{ in} \\
& \quad \text{let } \Gamma_b^\sharp = [\![C_b]\!]^\sharp(\Gamma_s^\sharp) \text{ in} \\
& \quad \text{let } D_e = \{|x \leftarrow x_e|\}(\Gamma_b^\sharp[x_e]) \text{ in} \\
& \quad \text{let } D'_s = D_i \sqcup \texttt{killall}(D_e) \text{ in} \\
& \quad \text{if } (D'_s \sqsubseteq D_s) \text{ then } D'_s \text{ else } \texttt{fix}(D_s \nabla D'_s) \\
& \text{in} \\
& \text{let } D' = \texttt{fix}(D_i) \text{ in} \\
& [\![C]\!]^\sharp(\Gamma^\sharp[x \mapsto D'])
\end{array}
$$

Fig. 8: Algorithm: Evaluating LAF terms with usual abstract domains. $\{|x \leftarrow expr|\}$ denotes the transfer function for the assignment $x \leftarrow e$.

C Translation to SMT and Horn

C.1 Translation to SMT

We begin by this translation, as it is easier

Intuitively, the translation associates to each LAF variable x a pair of SMT variables $\langle c_x, v_x \rangle$, where

- c represents the necessary condition for x to be different from $\bot$
- v represents the value to which x evaluates.

Thus if x has value $\bot$, then c_x has value $false$ and v_x can be anything; if x has value 33, then c_x is $true$ and v_x is also 33.

More formally, the translation $(\!|\cdot|\!)$ creates a first-order formula φ and a mapping $\mathcal{M}$ from LAF variables x to SMT variables $\langle c_x, v_x\rangle$; such that if $x \Downarrow u$, then the formula $\varphi \wedge c_x \wedge v_x = u$ is satisfiable. The converse is not true, because the translation of loops is over-approximated, as done in weakest-precondition computation. This is "fixed" by generating Horn clauses instead of SMT, which extends this translation to handle loops.

$$\textsc{Empty and Result}$$
$$(\!|[]|\!)(\mathcal{M}, \varphi) = \langle \mathcal{M}, \varphi \rangle$$

$$\textsc{Theory op}$$
$$\frac{\mathcal{M}[x_1] = \langle c_1, v_1\rangle \quad \ldots \quad \mathcal{M}[x_n] = \langle c_n, v_n\rangle \quad c \text{ fresh} \quad v \text{ fresh}}{\begin{array}{c}(\!|\text{let } x = op_n(x_1, \ldots, x_n) \text{ in } C|\!)(\mathcal{M}, \varphi) = \\ (\!|C|\!)(\mathcal{M}[x \mapsto \langle c, v\rangle],\ \varphi \wedge (c = c_1 \wedge \ldots \wedge c_n) \wedge (v = op_n(v_1, \ldots, v_n)))\end{array}}$$

$$\textsc{Unknown}$$
$$\frac{v \text{ fresh}}{(\!|\text{let } x = \text{unknown}_{\mathcal{S}}() \text{ in } C|\!)(\mathcal{M}, \varphi) = (\!|C|\!)(\mathcal{M}[x \mapsto \langle true, v\rangle], \varphi)}$$

$$\textsc{Assume}$$
$$\frac{\mathcal{M}[x_1] = \langle c_1, v_1\rangle \quad \mathcal{M}[x_2] = \langle c_2, v_2\rangle \quad c \text{ fresh} \quad v \text{ fresh}}{\begin{array}{c}(\!|\text{let } x = \text{assume}(x_1, x_2) \text{ in } C|\!)(\mathcal{M}, \varphi)\ = \\ (\!|C|\!)(\mathcal{M}[x \mapsto \langle c, v\rangle], \varphi \wedge (c = c_1 \wedge c_2 \wedge v_1) \wedge (v = v_2))\end{array}}$$

$$\textsc{Nondet}$$
$$\frac{\mathcal{M}[x_1] = \langle c_1, v_1\rangle \quad \mathcal{M}[x_2] = \langle c_2, v_2\rangle \quad c \text{ fresh} \quad v \text{ fresh}}{\begin{array}{c}(\!|\text{let } x = \text{nondet}(x_1, x_2) \text{ in } C|\!)(\mathcal{M}, \varphi) = \\ (\!|C|\!)(\mathcal{M}[x \mapsto \langle c, v\rangle],\ \varphi \wedge \big(c \Rightarrow ((c_1 \wedge v = v_1) \vee (c_2 \wedge v = v_2))\big) \wedge (c = c_1 \vee c_2))\end{array}}$$

$$\textsc{Mu}$$
$$\frac{\mathcal{M}[x_0] = \langle c_0, v_0\rangle \quad v \text{ fresh}}{(\!|\text{let } x = (\mu x_s.\ t_b[x_e])(x_i) \text{ in } C|\!)(\mathcal{M}, \varphi) = (\!|C|\!)(\mathcal{M}[x \mapsto \langle c_0, v\rangle], \varphi)}$$

Theorem 7. *Let $\langle \varphi, \mathcal{M}\rangle = (\!|C|\!)$. Let $\Gamma \in [\![C]\!]$. Let x be a variable of C. Then, x is in $\mathcal{M}$ and we choose $\langle c_x, v_x\rangle = \mathcal{M}[x]$. Then we have:*

$$\begin{cases} \exists \Gamma \in [\![C]\!](\varepsilon) : \Gamma[x] = \bot & \Rightarrow \quad \varphi \wedge \neg c_x \text{ is satisfiable} \\ \exists \Gamma \in [\![C]\!](\varepsilon) : \Gamma[x] = u \neq \bot & \Rightarrow \quad \varphi \wedge c_x \wedge (v_x = u) \text{ is satisfiable} \end{cases}$$

Proof. The proof is by induction: at each step of the algorithm, the $\varphi, \mathcal{M}$ produced verify the property for the variables already translated (which are in $\mathcal{M}$). For each construct, we show how a model of the formula $\mathscr{M}$ for φ can be extended to also satisfy the new constraints.

Corollary 1. *If $\varphi \wedge c_x \wedge (v_x = u)$ is unsatisfiable, then $\forall \Gamma \in [\![C]\!](\varepsilon)$, we have $\Gamma[x] \neq u$*

Proof. This is just the contrapposite.

Thus in practice this translation allows to prove that a variable can never be equal to some value. In particular in the case of boolean values, it can be used to prove that some condition can never be false, i.e. is always true; this allows to prove assertions about the program.

Note that the translation is linear in the size of the term, notably because we create new SMT variable for every LAF variable. Because the translation is linear, we can see the translation to LAF term + conversion to SMT formula as another implementation of the efficient weakest precondition technique of Leino[21].

C.2 Translation to Horn

The translation is relatively similar, except that we make use of Horn clauses to handle the recursion in the μ term. The only real issue is that in LAF term, the body of the μ can use variables defined outside of the body; this is called environment capture in functional language. This is fixed by explicitly passing the contents of these captured variables in the translation of the body.

D Translation from WHILE

The translation is relatively standard. The memory state corresponding to each statement is represented by the tuple of the values of each variable; $(\!|\cdot|\!)$ is the constant mapping from variables to indices. We have used "$[M$ except $i \mapsto x]$" as a syntactic sugar for "$\langle M[0], M[1], \ldots, M[i-1], x, M[i+1], \ldots, M[n]\rangle$". The translation is quite naive; in particular the operations on tuples get/set, and *nondet* of tuples, should be simplified.

$$
\begin{aligned}
(\!|\texttt{var}|\!)(C, M) &= \langle C[\text{let } x' = M[(\!|var|\!)]], x' \rangle \\
(\!|\text{op}_n(e_1, \ldots, e_n)|\!)(C, M) &= \text{let } \langle C_1, x_1 \rangle = (\!|e_1|\!)(C, M) \text{ in} \\
&\quad \text{let } \langle C_2, x_2 \rangle = (\!|e_2|\!)(C_1, M_1) \text{ in} \\
&\quad \vdots \\
&\quad \text{let } \langle C_n, x_n \rangle = (\!|e_n|\!)(C_{n-1}, M_{n-1}) \text{ in} \\
&\quad \langle C[\text{let } x' = op_n(x_1, \ldots, x_n)], x' \rangle \\
\{\!|\texttt{var}:= e|\!\}(C, M) &= \text{let } \langle C', x \rangle = (\!|e|\!)(C, M) \text{ in} \\
&\quad \langle C', [M \text{ except } (\!|\texttt{var}|\!) \mapsto \texttt{x}] \rangle \\
\{\!|s_1; s_2|\!\}(C, M) &= \text{let } \langle C_1, M_1 \rangle = \{\!|s_1|\!\}(C, M) \text{ in} \\
&\quad \{\!|s_2|\!\}(C_1, M_1) \\
\{\!|\texttt{if } e \texttt{ then } s_{\text{then}} \texttt{ else } s_{\text{else}}|\!\}(C, M) &= \text{let } \langle C_1, x \rangle = (\!|e|\!)(C, M) \text{ in} \\
&\quad \text{let } C_2 = C_1[\text{let } nx = \neg x] \text{ in} \\
&\quad \text{let } \langle C_3, M_{\text{then}} \rangle = \{\!|s_{\text{then}}|\!\}(C_2, M) \text{ in} \\
&\quad \text{let } \langle C_4, M_{\text{else}} \rangle = \{\!|s_{\text{else}}|\!\}(C_3, M) \text{ in} \\
&\quad \text{let } C_5 = C_4[\text{let } M'_{\text{then}} = \text{assume}(x, M_{\text{then}})] \text{ in} \\
&\quad \text{let } C_6 = C_5[\text{let } M'_{\text{else}} = \text{assume}(nx, M_{\text{else}})] \text{ in} \\
&\quad \text{let } C_7 = C_6[\text{let } M' = \text{nondet}(M'_{\text{then}}, M'_{\text{else}})] \text{ in} \\
&\quad \langle C_7, M' \rangle \\
\{\!|\texttt{while } e \texttt{ do } s \texttt{ done}|\!\}(C, M) &= \text{let } \mathbf{M} = \text{fresh}() \text{ in} \\
&\quad \text{let } \langle C_1, x \rangle = (\!|e|\!)(C, \mathbf{M}) \text{ in} \\
&\quad \text{let } C_2 = C_1[\text{let } M_2 = \text{assume}(x, \mathbf{M})] \text{ in} \\
&\quad \text{let } \langle C_3, M_3 \rangle = \{\!|s|\!\}(C_2, M_2) \text{ in} \\
&\quad \text{let } C_4 = C[\text{let } M' = (\mu \mathbf{M}.C_3[M_3])(M) \text{ in} \\
&\quad \text{let } \langle C_5, x' \rangle = (\!|e|\!)(C_4, M') \text{ in} \\
&\quad \text{let } C_6 = C_5[\text{let } nx' = \neg x'] \text{ in} \\
&\quad \text{let } C_7 = C_6[\text{let } M'' = \text{assume}(nx', M')] \text{ in} \\
&\quad \langle C_7, M'' \rangle
\end{aligned}
$$

Fig. 9: Translation from the WHILE language to LAF terms

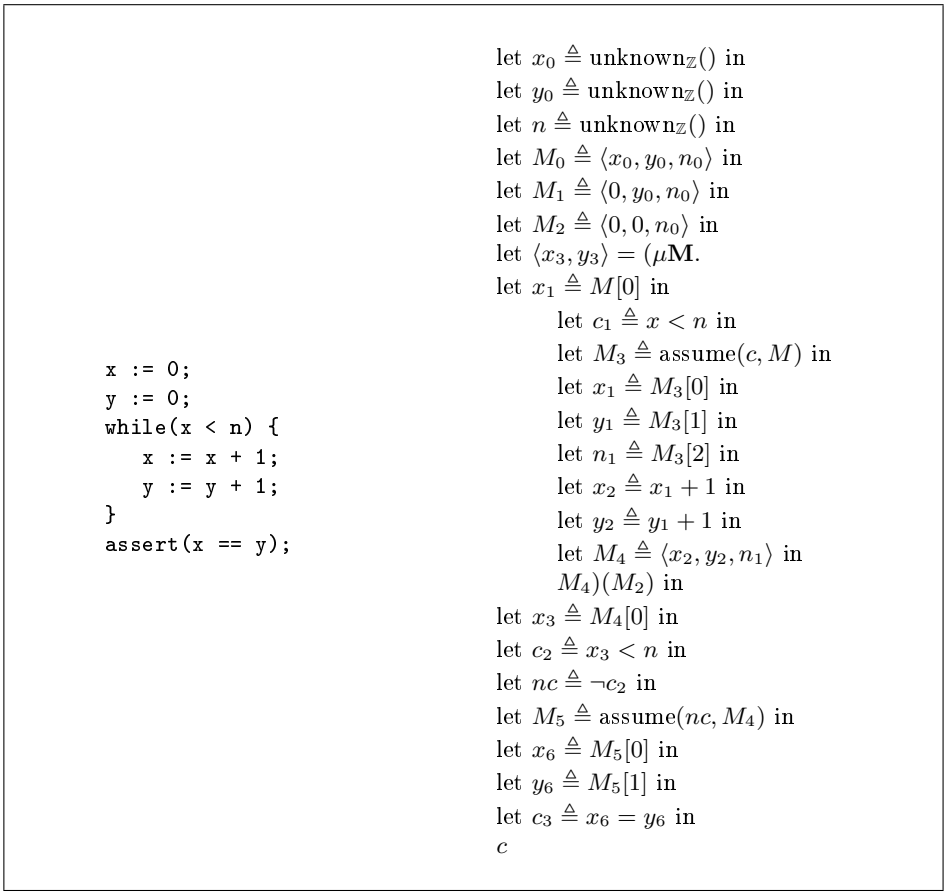

Fig. 10: Full (naive) translation of the assertion in the program on the left. A better translation would remove useless terms such as M_1 and x_0, and would realize that n is not modified in the loop (and thus does not need to be in the loop tuple).