An Abstract Domain of Uninterpreted Functions

Graeme Gange[1], Jorge A. Navas[2], Peter Schachte[1], Harald Søndergaard[1], and Peter J. Stuckey[1]

[1] Department of Computing and Information Systems,
The University of Melbourne, Victoria 3010, Australia
{gkgange,schachte,harald,pstuckey}@unimelb.edu.au
[2] NASA Ames Research Center, Moffett Field, CA 94035, USA
jorge.a.navaslaserna@nasa.gov

Abstract. We revisit relational static analysis of numeric variables. Such analyses face two difficulties. First, even inexpensive relational domains scale too poorly to be practical for large code-bases. Second, to remain tractable they have extremely coarse handling of non-linear relations. In this paper, we introduce the subterm domain, a weakly relational abstract domain for inferring equivalences amongst sub-expressions, based on the theory of uninterpreted functions. This provides an extremely cheap approach for enriching non-relational domains with relational information, and enhances precision of both relational and non-relational domains in the presence of non-linear operations. We evaluate the idea in the context of the software verification tool SeaHorn.

1 Introduction

This paper investigates a new approach to relational analysis. Our aim is to develop a method that scales to very large code bases, yet maintains a reasonable degree of precision, also for programs that use non-linear numeric operations.

Abstract interpretation is the well-established theoretical framework for sound reasoning about program properties. It provides a means for comparing program analyses, especially with respect to the granularity of information (precision) that analyses allow us to statically extract from programs. On the whole, reducing such questions to questions about *abstract domains*. An abstract domain, essentially, specifies the (limited) language of judgements we are able to use when reasoning statically about a program's runtime behaviour.

A class of abstract domains that has received particular attention are the *numeric* domains—those supporting reasoning about variables of numeric (often integer or rational) type. Numeric domains are important because of the numerous applications in termination and safety analyses, such as overflow detection and out-of-bounds array analysis. The *polyhedral* abstract domain [9] allows us to express linear arithmetic constraints (equalities and inequalities) over program state spaces of arbitrary finite dimension k. But high expressiveness comes at a cost; analysis using the polyhedral domain does not scale well to large code bases. For this reason, a number of abstract domains have been proposed, seeking to strike a better balance between cost and expressiveness.

Language restriction. The primary way of doing this is to limit expressiveness, that is, to restrict the language of allowed judgements. Most commonly this is done by expressing only 1- or 2-dimensional projections of the program's (abstract) state space, often banning all but a limited set of coefficients in linear constraints. Examples of this kind of restriction to polyhedral analysis abound, including zones [19], TVPI [23, 22], octagons [20], pentagons [18], and logahedra [14]. These avoid the exponential behaviour of polyhedra, instead offering polynomial (typically quadratic or cubic) decision and normalization procedures. Still, they have been observed to be too expensive in practice for industrial codebases [18, 24]. Hence other "restrictive" techniques have been proposed which are sometimes integral to an analysis, sometimes orthogonal.

Dimensionality restriction. These methods aim to lower the dimension k of the program (abstract) state space, by replacing the full space with several lower-dimension subspaces. Variables are separated into "buckets" or *packs* according to some criterion. Usually the packs are disjoint, and relations can be explored only amongst variables in the same pack (relaxations of this have also been proposed [4]). The criterion for pack membership may be syntactic [8] or determined dynamically [24]. A variant is to only permit relations between sets; in the Gauge domain [25], relations are only maintained between program variables and introduced loop counters, not between sets of program variables.

Closure restriction. Some methods abandon the systematic transitive closure of relations (and therefore lack a normal form for constraints). Constraints that follow by transitive closure may be discovered lazily, or not at all. Closure restriction was used successfully with the pentagon domain; a tolerable loss of precision was compensated for by a significant cost reduction [18].

All of the work discussed up to this point has, in some sense, started from an ideal (polyhedral) analysis and applied restrictions to the degree of "relationality." A different line of work starts from very basic analyses and adds mechanisms to capture relational information. These approaches do not focus on restrictions, but rather on how to compensate for limited precision using "symbolic" reasoning. Such symbolic methods maintain selected syntactic information about computations and use this to enhance precision. The primary examples are Miné's *linearization* method [21], based on "symbolic constant propagation" and Chang and Leino's congruence closure extension [5].

Polyhedral analysis and its restrictions tend to fall back on overly coarse approximation when faced with non-linear operations such as multiplication, modulus, or bitwise operations. Higher precision is desirable, assuming the associated cost is limited. Consider the example shown in Figure 1(a). Figure 2(a) shows the possible program states when execution reaches point `A`. With octagons, the strongest claim that can be made at that point is

$$0 \leq x \leq 10, -10 \leq y \leq 10, y - z \leq 90, z - y \leq 90, x + z \geq -90, z - x \leq 90$$

```
    x = nondet(0,10)
    y = nondet(-10,10)
    z = x*y
A:
    if (y < 0) {
        z = -z
    }
B:
```

(a)

```
    u = nondet(0,10)
    v = nondet(0,10)
    w = nondet(0,10)
    if (*)
        t = u + v  else t = u + w
    if (t < 3)
        u = u + 3  else u = 3
C:
```

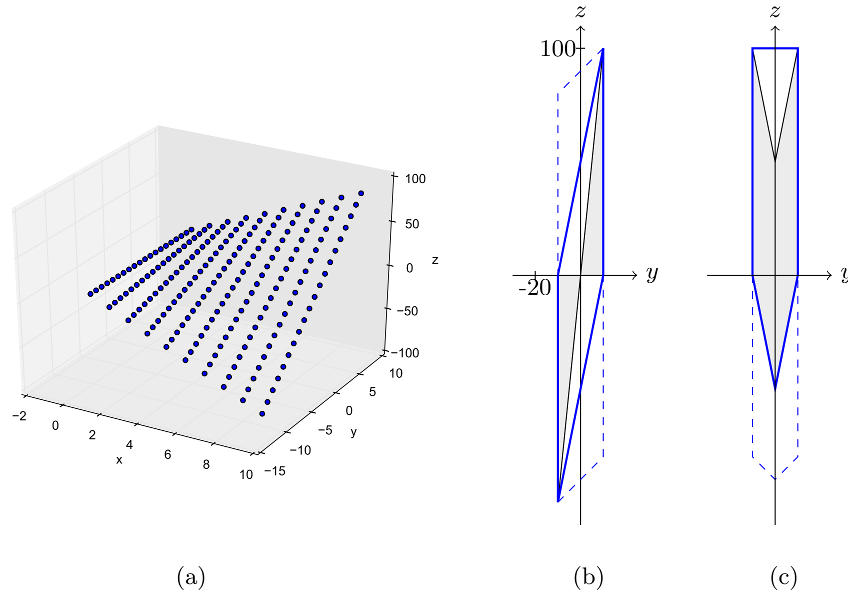
(b)

Fig. 1: Two example programs

Figure 2(b) shows the projection on the y-z plane. Almost all interaction between y and z has been lost and as a result, we fail to detect that z is non-negative at point `B`. The best possible polyhedral approximation adds

$$z \geq -10x, z \geq 10x + 10y - 100, z \leq 10x, z \leq 100 - 10x + 10y$$

While this expresses more of the relationship between x, y and z, we can still only infer $z \geq -50$ at point `B`.



(a) (b) (c)

Fig. 2: (a) The reachable states at point `A` in Figure 1(a); (b) the result of polyhedral analysis at point `A`, projected onto the y-z plane, assuming analysis performs case split on the sign of y (the convex hull forming a lozenge); (c) the result of polyhedral analysis at point `B`. Dashed lines show octagon invariants.

Octagons	$y - z \leq 90, z - y \leq 90,$ $x + z \geq -90, z - x \leq 90$	$0 \leq t, 3 \leq u \leq 20, u + t \leq 23$
Polyhedra	$z \geq -10x, z \geq 10x + 10y - 100,$ $z \leq 10x, z \leq 100 - 10x + 10y$	$0 \leq t \leq 20, 3 \leq u \leq 5$
Subterms	$0 \leq z \leq 100$	$0 \leq t \leq 20, 3 \leq u \leq 5$

Table 1: States inferred for Figure 1's programs, points `B` (left) and `C` (right)

In practice, weaker results may well be produced. A commonly used octagon library yields $y \in [-10, 10], z \in [-100, 100]$, rather than the dashed projections shown in Figure 2(b) and (c). For polyhedral analysis, multiplication is often handled by projection and case-splitting. The two grey triangles in Figure 2(b) show the result, at point `A`, of case analysis according to the sign of y, as projected onto the y-z plane; the lozenge is the convex hull. This explains how a commonly used library infers $\{z \geq 5y - 50, z \leq 5y + 50\}$ at point `A`. The pen-nib shaped area in Figure 2(c) shows the result, at point `B`, of polyhedral analysis. Note that the triangle below the y axis is in fact infeasible.

Contribution. The proposal presented in this paper differs from all of the above. It combines closure restriction and a novel symbolic approach. We extract and utilise *shared expression* information to improve the precision of cheap non-relational analyses (for example, interval analysis), at a small added cost. The idea is to treat the arithmetic operators as uninterpreted function symbols. This allows us to replace expensive convex hull operations by a combination of constraint propagation and term anti-unification. The resulting *subterm domain* $\mathcal{ST}$ is an abstract domain of syntactic equivalences. It can be used to augment non-relational domains with relational information, and to improve precision of (possibly relational) domains in the presence of complex operations. The improvement is not restricted to non-linear operations; it can equally well support weakly relational domains that are unable to handle large coefficients.

Table 1 summarises the analysis results for the two programs in Figure 1, compared with the results of (ideal) octagon and polyhedral analysis.[3] Note how the subterm domain obtains a tight lower bound on z as well as a tight upper bound on u.

The method has been implemented, and the experiments described later in this paper suggest the combination strikes a happy balance between precision and cost. After Section 2's preliminaries, Section 3 provides algorithms for operations on systems of terms, and Section 4 shows how this can be used to enhance a numeric domain. Section 5 provides comparison with the closest related work. Section 6 reports on experimental results and Section 7 concludes.

[3] We show transitive reductions and omit trivial bounds for variables. The result obtained by the subterm domain for `C`, includes, behind the scenes, a *term equation* $t = u + s$ and a bound $0 \leq s \leq 10$ on the freshly introduced variable s.

2 Preliminaries

Abstract interpretation. In standard abstract interpretation, a concrete domain $C^\natural$ and its abstraction $C^\#$ are related by a *Galois connection* (α, γ), consisting of an abstraction function $\alpha : C^\natural \mapsto C^\#$ and concretization function $\gamma : C^\# \to C^\natural$. The best approximation of a function $f^\natural$ on $C^\natural$ is $f^\#(\varphi) = \alpha(f^\natural(\gamma(\varphi)))$. When analysing imperative programs, $C^\natural$ is typically the powerset of program states, and the corresponding lattice operations are $(\subseteq, \cup, \cap)$.

In a non-relational (or *independent attribute*) domain, the abstract state is either the bottom value $\bot_D$ (denoting an infeasible state), or a separate non-$\bot$ abstraction $x^\#$ for each variable x in some domain D_V (where each variable admits some feasible value). That is, $D = \{\bot_D\} \cup (D_V \setminus \{\bot_D\})^{|V|}$.

Sometimes *backwards* reasoning is required, to infer the set of states which may/must give rise to some property. The *pre-image* transformer $F_{\mathcal{D}}^{-1}([\![\mathtt{S}]\!])(\varphi)$ yields φ_{pre} such that $(F_{\mathcal{D}}([\![\mathtt{S}]\!])(\varphi') = \varphi) \Rightarrow (\varphi' \sqsubseteq \varphi_{pre})$. Finding the minimal pre-image of a complex (non-linear) operation can be quite expensive, so pre-image transformers provided by numeric domains are usually coarse approximations.

We shall sometimes need to *rename* abstract values. Given a binary relation $\pi \subseteq V \times V'$ and an element φ of an independent attribute domain over V, the renaming $\pi(\varphi)$ is given by:

$$rename_\pi(\varphi) = \{x' \mapsto \bigsqcap_{\mathcal{D}\,(x,x') \in \pi} \varphi(x) \mid x' \in image(\varphi)\}$$

The corresponding operation is more involved for relational domains. Assuming $\mathcal{D}$ is closed under existential quantification, $\mathcal{D}$ can maintain systems of equalities and V and V' are disjoint, we have $rename_\pi(\varphi) = \exists V.\, (\varphi \sqcap \{x = x' \mid (x, x') \in \pi\})$.

Term equations. The set $\mathcal{T}$ of *terms* is defined recursively: every term is either a variable $v \in TVar$ or a construction $F(t_1, \ldots, t_n)$, where $F \in Fun$ has arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms. A *substitution* is an almost-identity mapping $\theta \in TVar \to \mathcal{T}$, naturally extended to $\mathcal{T} \to \mathcal{T}$. We use standard notation for substitutions; for example, $\{x \mapsto t\}$ is the substitution θ such that $\theta(x) = t$ and $\theta(v) = v$ for all $v \neq x$. Any term $\theta(t)$ is an *instance* of term t.

If we define $t \sqsubseteq t'$ iff $t = \theta(t')$ for some substitution θ then $\sqsubseteq$ is a preorder. Define $t \equiv t'$ iff $t \sqsubseteq t' \wedge t' \sqsubseteq t$. The set $\mathcal{T}_{/\equiv} \cup \{\bot\}$, that is $\mathcal{T}$ partitioned into equivalence classes by $\equiv$ plus $\{\bot\}$, is known to form a complete lattice, the so-called *term lattice*.[4] A *unifier* of $t, t' \in \mathcal{T}$ is an idempotent substitution θ such that $\theta(t) = \theta(t')$. A unifier θ of t and t' is a *most general unifier* of t and t' iff $\theta' = \theta' \circ \theta$ for every unifier θ' of t and t'.

If we can calculate most general unifiers then we can find meets in the term lattice: if θ is a most general unifier of t and t' then $\theta(t)$ is the most general term that simultaneously is an instance of t and an instance of t', so $\theta(t)$ is the meet of t and t'. Similarly, the join of t and t' is the *most specific generalization*; algorithms are available that calculate most specific generalizations [15].

[4] $\sqsubseteq$ is extended to the term lattice by defining $\bot \sqsubseteq t$ for all elements $t \in \mathcal{T}_{/\equiv}$.

Given a set of terms $S \subseteq \mathcal{T}$ and equivalences $E \subseteq (S \times S)$, we can partition S into equivalent terms. Terms t and s are *equivalent* ($t \equiv s$) if they are identical constants, are *deemed* equal, or $t = f(t_1, \ldots, t_m)$ and $s = f(s_1, \ldots, s_m)$ such that for all i, $t_i \equiv s_i$. Finding this partitioning is the well-studied *congruence closure* problem, of complexity $\mathrm{O}(|S| \log |S|)$ [10]. Of relevance is the case $|E| = 1$ (introduction of a single equivalence), which can be handled in $\mathrm{O}(|S|)$ time.

In the following, it will be necessary to distinguish a term as an object from the syntactic expression it represents. We shall use $\mathbf{id}(t)$ to denote the *name* of a term, and $\mathbf{def}(t)$ to denote the expression.

3 The Subterm Domain $\mathcal{ST}$

An element of the subterm domain consists of a mapping $\eta : V \mapsto \mathcal{T}$ of program variables to terms. While the domain structure derives from uninterpreted functions, we must reason about the corresponding concrete computations. We accordingly assume each function symbol F has been given a semantic function $\mathbb{S}(F) : \mathcal{S}^n \to \mathcal{S}$. Given some assignment $\theta : TVar \to \mathcal{S}$ of *term variables* to scalar values, we can then recursively define the evaluation $\mathbb{E}(t, \theta)$ of a term under θ.

$$\mathbb{E}(x, \theta) = \theta(x)$$
$$\mathbb{E}(f(t_1, \ldots, t_n), \theta) = \mathbb{S}(f)(\mathbb{E}(t_1, \theta), \ldots, \mathbb{E}(t_n, \theta))$$

We say a concrete state $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ *satisfies* mapping η iff there is an assignment θ of values to term variables such that for all x_i, $\mathbb{E}(\eta(x_i), \theta) = v_i$. The concretization $\gamma(\eta)$ is the set of concrete states which satisfy η.

However, the syntactic nature of our domain gives us difficulties. While we can safely conclude that two (sub-)terms are equivalent, we have no way to conclude that two terms differ. No Galois connection exists for this domain; multiple sets of definitions could correspond to a given concrete state. Even if states η_1 and η_2 are both valid approximations of the concrete state, the same does not necessarily hold for $\eta_1 \sqcap \eta_2$.

Example 1. Consider two abstract states:

$$\{x \mapsto +(a_1, 7), y \mapsto a_1, z \mapsto a_2\} \qquad \{x \mapsto +(3, b_1), y \mapsto b_2, z \mapsto b_1\}$$

These correspond to the sets of states satisfying $x = y + 7$ and $x = 3 + z$ respectively. Many concrete states satisfy both approximations; one is $(x, y, z) = (7, 0, 4)$. However, a naive application of unification would attempt to unify $+(y, 7)$ with $+(3, z)$, which would result in unifying y with 3, and z with 7.

Cousot and Cousot [7] discuss the consequences of a missing best approximation, and propose several approaches for repair: strengthening or weakening the domain, or nominating a best approximation through a widening/narrowing. However, these are of limited value in our application. Strengthening or weakening the domain enough that a best approximation is restored would greatly affect the performance or precision, and explicitly reasoning over the set of equivalent states is impractical. Using a widening/narrowing is sound advice, but offers minimal practical guidance.

$$F_{\mathcal{ST}}[\![\mathtt{x} := \mathtt{f}(y_1, \ldots, y_n)]\!](\eta) = \eta[x \mapsto \mathtt{f}(\eta(y_1), \ldots, \eta(y_n))]$$

$$\eta_1 \sqcup \eta_2 = \{x \mapsto \mathsf{generalize}(\eta_1(x), \eta_2(x)) \mid x \in V\}$$
$$\mathsf{generalize}(\mathsf{c}, \mathsf{c}) = \mathsf{c}$$
$$\mathsf{generalize}(f(t_1, \ldots, t_n), f(s_1, \ldots, s_n)) = f(u_1, \ldots, u_n)$$
$$\quad \textbf{where } u_i = \mathsf{generalize}(t_i, s_i)$$
$$\mathsf{generalize}(X, Y) = \mathtt{freshvar}$$

Fig. 3: Definitions of variable assignment and $\sqcup$ in $\mathcal{ST}$.

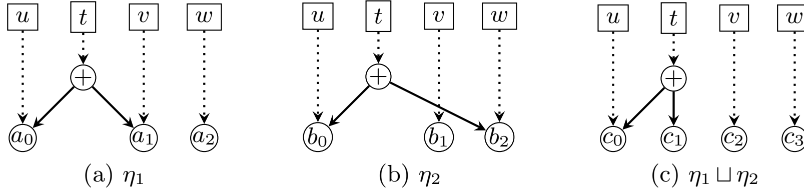


Fig. 4: State at the end of the first (a) *then* and (b) *else* branches in Figure 1(b), and (c) the join of the two states.

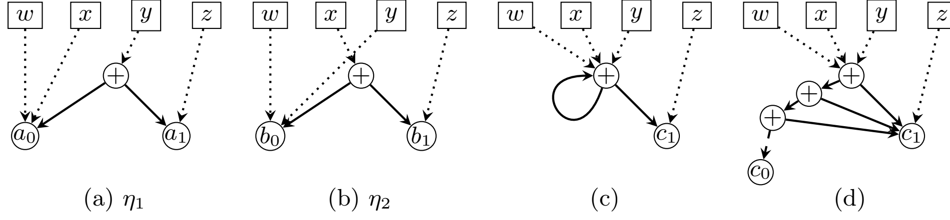
3.1 Operations on $\mathcal{ST}$

We must now specify several operations: state transformers for program statements, join, meet, and widening. Assignment, join and widening all behave nicely under $\mathcal{ST}$; meet is discussed in Section 3.2.

Figure 3 shows assignment and join operations on $\mathcal{T}$. Note that calls to generalize are cached, so two calls to generalize(s,t) will return the same term variable. In the case of $\mathcal{ST}$, the join on the lattice is safe: as $\eta_1 \sqsubseteq \eta_2 \Rightarrow \gamma(\eta_1) \sqsubseteq_{\mathcal{C}^\natural} \gamma(\eta_2)$ and $\sqcup$ and $\sqcup_{\mathcal{C}^\natural}$ are least upper bounds on their respective domains, we have $\gamma(\eta_1) \sqsubseteq \gamma(\eta_1 \sqcup \eta_2)$ and $\gamma(\eta_2) \sqsubseteq \gamma(\eta_1 \sqcup \eta_2)$, so $\gamma(\eta_1) \sqcup_{\mathcal{C}^\natural} \gamma(\eta_2) \sqsubseteq_{\mathcal{C}^\natural} \gamma(\eta_1 \sqcup \eta_2)$. The worst-case complexity of the join is $\mathrm{O}(|\eta_1||\eta_2|)$. However, typical behaviour should be expected to be closer to linear, as most shared terms are either shared in both (so are only considered once) or are trivially distinct (and are replaced by a variable). This is borne out in experiments, see Section 6.

Every term in $\eta_1 \sqcup \eta_2$ corresponds to some specialization in η_1 and η_2. We shall use $\pi^{\eta_1 \mapsto \eta_1 \sqcup \eta_2}$ to denote the relation that maps terms in η_1 to corresponding terms in $\eta_1 \sqcup \eta_2$.

Example 2. Consider again Figure 1(b). At the exit of the first if-then-else, we get term-graphs η_1 and η_2 shown in Figure 4(a) and 4(b). For $\eta_1 \sqcup \eta_2$, we first compute the generalization of $\eta_1(u) = a_0$ with $\eta_2(u) = b_0$, obtaining a fresh variable c_0. $\eta_1(t)$ and $\eta_2(t)$ are both $(+_2)$ terms, so we recurse on the children; the generalization of (a_0, b_0) has already been computed, so we re-use the existing variable; but we must allocate a fresh variable for (a_1, b_2), resulting in t being mapped to $(+)(c_0, c_1)$. We repeat this process for v and w, yielding the state shown in Figure 4(c). Note that the result captures the fact that in both branches, t is computed by adding some value to u.



(a) η_1 (b) η_2 (c) (d)

Fig. 5: Abstract states η_1 and η_2, whose conjunction $\eta_1 \wedge \eta_2$ (c) cannot be represented in $\mathcal{ST}$; it has an infinite descending chain of approximations (d).

3.2 The quasi-meet $\widetilde{\sqcap}$

We require our quasi-meet $\sqcap_{\mathcal{ST}}$ to be a sound approximation of the concrete meet, that is, $\gamma(\eta_1) \sqcap_{\mathcal{C}^\natural} \gamma(\eta_2) \sqsubseteq_{\mathcal{C}^\natural} \gamma(\eta_1 \sqcap_{\mathcal{ST}} \eta_2)$. Ideally, we would like to preserve several other properties enjoyed by lattice operations:

Minimality: If $\eta_1 \sqsubseteq_{\mathcal{ST}} \eta_2$, then $(\eta_1 \sqcap_{\mathcal{ST}} \eta_2) = \eta_1$
Monotonicity: If $\eta_1 \sqsubseteq_{\mathcal{ST}} \eta_1'$, then $(\eta_1 \sqcap_{\mathcal{ST}} \eta_2) \sqsubseteq_{\mathcal{ST}} (\eta_1' \sqcap_{\mathcal{ST}} \eta_2)$

These are important for precision and termination respectively. However, in the absence of a unique greatest lower bound these properties are mutually exclusive, so the quasi-meet must be handled carefully to avoid non-termination [12].

A simple quasi-meet (denoted by $\widetilde{\sqcap}$, as distinct from a 'true' meet $\sqcap$) is to adopt the approach of [21], deterministically selecting the term for each variable from either η_1 or η_2. Minimality can be achieved by selecting the more precise term (according to $\sqsubseteq_{\mathcal{ST}}$) when several choices exist. However, this discards a great deal of information present in the conjunction. Of particular concern is the loss of variable equivalences which are implied by $\eta_1 \wedge \eta_2$ (the *logical conjunction* of η_1 and η_2), but not by η_1 and η_2 individually.

We can infer all sub-term (and variable) equivalences of $\eta_1 \wedge \eta_2$ using the congruence closure algorithm. Unfortunately, not only may this yield multiple incompatible definitions for a variable, the resulting definitions may be cyclic.

Example 3. Consider the abstract states η_1, η_2 shown in Figure 5(a) and 5(b). Computing $\eta_1 \wedge \eta_2$, we start with constraints $\{\eta_1(v) = \eta_2(v) \mid v \in \{w, x, y, z\}\}$:

$$\{t = (+)(a_0, a_1)\} \cup \{s = (+)(b_0, b_1)\} \cup \{a_0 = b_0, a_0 = s, t = b_0, a_1 = b_1\}$$

After congruence closure, the terms are split into two equivalence classes:

$$E_1 = \{a_0, b_0, s, t\}, E_2 = \{a_1, b_1\}$$

We then wish to extract an element of $\mathcal{ST}$ which preserves as much of this information as possible. This conjunction, shown in Figure 5(c), cannot be precisely represented in $\mathcal{ST}$ – Figure 5(d) gives an infinite descending chain of approximations. Note that we could obtain incomparable elements of $\mathcal{ST}$ by pointing each of $\{w, x, y\}$ at different $(+)$ nodes in 5(d).

```
quasi-meet(η1, η2)
  % Partition terms into congruence classes
  Eq := congruence-close(Defs(η1) ∪ Defs(η2) ∪ {η1(x) = η2(x) | x ∈ V})
  for each e ∈ Eq
    indegree(e) := |{x | η1(x) ∈ eq}|
  stack := ∅, repr := ∅, tvar := ∅
  for each x ∈ V
    η(x) := build-repr(Eq(η1(x)))
  return η

build-repr(eq)
  if eq ∈ stack % If this is a back-edge, break the cycle
    if eq ∉ tvar
      tvar(eq) := freshvar()
    return tvar(eq)
  if eq ∈ repr % If we have already computed the representative, return it
    return repr(eq)
  % The equivalence class has not yet been seen; select best concrete definition
  stack.push(eq)
  if terms(eq) = ∅ % No concrete definition exists
    r_eq := freshvar
  else
    f(s1, ..., sm) := argmax_{f(s1,...,sm) ∈ mem(eq)} Σ_i { 0                  if Eq(s_i) ∈ stack
                                                          { indegree(Eq(s_i))  otherwise
    for each i ∈ 1, ..., m % Construct the representative for each subterm
      r_i := build-repr(Eq(s_i))
    r_eq := f(r1, ..., rm)
  repr(eq) := r_eq
  stack.pop(eq)
  return r_eq
```

Fig. 6: Algorithm to compute $\widetilde{\sqcap}_{\mathcal{ST}}$. *Eq*, *stack*, *repr*, *tvar* and *indegree* are global.

We therefore need a strategy for choosing a finite approximation of $\eta_1 \wedge \eta_2$ in $\mathcal{ST}$. There are two elements to this decision: how a representative for each equivalence class is chosen, and how cycles are broken. We wish to preserve as many equivalences as possible, particularly between variables.

The algorithm for computing $\eta_1 \,\widetilde{\sqcap}_{\mathcal{ST}}\, \eta_2$ is given in Figure 6. We first partition the terms in $\eta_1 \cup \eta_2$ into equivalence classes using the congruence closure algorithm, then count the external references to each class. These counts, recorded in *indegree*, give us an indication of how valuable each class is, to discriminate between candidate representatives. $Eq(t)$ returns the equivalence class containing term t, and $mem(eq)$ denotes the set of *non-variable* terms in class *eq*.

We then progressively construct the resulting system of terms, starting from the mapping of each variable. Each equivalence class *eq* corresponds to at most

two terms in the meet; the main representative $repr(eq)$, and a term variable $tvar(eq)$. Instantiating a term $f(s_1, \ldots, s_m)$, we look-up the corresponding equivalence class $eq_i = Eq(s_i)$, and check whether expanding its definition $repr(eq_i)$ (which may not yet be fully instantiated) would introduce a cycle. We then replace s_i with either the recursively constructed representative of eq_i (if the resulting system is acyclic), or the free variable $tvar(eq)$.

Example 4. Consider the abstract states η_1, η_2 shown in Figure 5. Congruence closure yields two equivalence classes: $q_1 = \{a_0, (+)(a_0, a_1), b_0, (+)(b_0, b_1)\}$, and $q_2 = \{a_1, b_1\}$. The construction of $\eta_1 \mathbin{\widetilde{\sqcap}} \eta_2$ starts with $Eq(w)$. We first mark q_1 as being on the stack to avoid cycles, then choose an appropriate definition to expand. The non-variable members of q_1 are $\{t_1 = (+)(a_0, a_1), t_2 = (+)(b_0, b_1)\}$. Both t_1 and t_2 have a single non-cycle incoming edge ($Eq(a_0) = Eq(b_0) = q_1$, which is already on the stack), so we arbitrarily choose t_1.

We must then expand the sub-terms of t_1. $Eq(a_0)$ is already on the stack, so cannot be expanded; this occurrence of a_0 is replaced with a fresh variable c_0. Now a_1 has no non-variable definitions, so a fresh variable c_1 is introduced. The stack then collapses, yielding $w \mapsto (+)(c_0, c_1)$.

The algorithm next considers x. A representative for q_1 has already been constructed, so x is mapped to $(+)(c_0, c_1)$, as is y. Finally, $Eq(z) = q_2$; this also has an existing representative, so c_1 is returned. The resulting abstract state is shown in Figure 7. □

The algorithm given in Figure 6 runs in $\mathrm{O}(n \log n)$ time, where $n = |\eta_1| + |\eta_2|$. The congruence closure step is run once, in $\mathrm{O}(n \log n)$ time. The main body of build-repr is run at most once per equivalence class. Computing and scoring the set of candidates is linear in $|eq|$, and happens once per equivalence class. We detect back-edges in constant time, by marking those equivalence classes which remain on the call stack – any edge to a marked class is a back-edge. So the reconstruction of η takes time $O(n)$ in the worst case. Therefore, the overall algorithm takes $\mathrm{O}(n \log n)$.

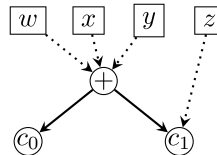


Fig. 7: $\eta_1 \mathbin{\widetilde{\sqcap}} \eta_2$

Note that $\eta_1 \mathbin{\widetilde{\sqcap}}_{\mathcal{ST}} \eta_2$ is sensitive to variable ordering, as this determines which sub-term occurrence is considered a back-edge, and thus not expanded.

As for $\sqcup_{\mathcal{ST}}$, each term in $\eta_1 \mathbin{\widetilde{\sqcap}}_{\mathcal{ST}} \eta_2$ corresponds to some set of terms in η_1 or η_2. As before, $\pi^{\eta_1 \mapsto \eta_1 \widetilde{\sqcap}_{\mathcal{ST}} \eta_2}$ denotes the mapping between terms in each operand and the result.

3.3 Logical assertions

Finally consider assertions $[\![\mathtt{x} \bowtie \mathtt{y}]\!]$, where $\bowtie \in \{=, \neq, <, \leq\}$. The abstract transformer for $[\![\mathtt{x} < \mathtt{y}]\!]$ and $[\![\mathtt{x} \leq \mathtt{y}]\!]$ is the identity function, as $\mathcal{ST}$ has no notion of inequalities. $\mathcal{ST}$ *can* infer information from a disequality $[\![\mathtt{x} \neq \mathtt{y}]\!]$, but only where η has already inferred equality between x and y:

$$F[\![\mathtt{x} \neq \mathtt{y}]\!]\eta = \begin{cases} \bot & \textbf{if } \eta(x) = \eta(y) \\ \eta & \textbf{otherwise} \end{cases}$$

In the case of an equality $[\![\mathtt{x} = \mathtt{y}]\!]$, we are left in a similar situation as for $\eta_1 \sqcap \eta_2$; we must reconcile the defining terms for x and y, plus any other inferred equivalences. This is done in the same way, by first computing equivalence classes, then extracting an acyclic system of terms. As we introduce only a single additional equivalence, we can use the specialized linear-time algorithm described in Section 3.4 of [10], then extract the resulting term system as for the meet.

4 $\mathcal{ST}$ as a Functor Domain

Assume we have some abstract domain $\mathcal{D}$ with the usual operations $\sqcap$, $\sqcup$, $F_{\mathcal{D}}$ and $F^{-1}_{\mathcal{D}}$ as described in Section 2. In the following, we assume $\mathcal{D}$ is not relational, so may only express independent properties of variables.

We would like to use $\mathcal{ST}$ to enhance the precision of analysis under $\mathcal{D}$. Essentially, we want a functor domain where $\mathcal{ST}$ is the functor instantiated with $\mathcal{D}$. While this is a simple formulation, it provides no path toward an efficient implementation. Where normally we use $\mathcal{D}$ to approximate the values of (or relationships between) variables in V, we can instead approximate the values of *terms* occurring in the program. An element of our lifted domain $\mathcal{ST}(\mathcal{D})$ is a pair $\langle \eta, \rho \rangle$ where η is a mapping of program variable to terms, and $\rho \in \mathcal{D}$ approximates the set of satisfying term assignments.

4.1 Operations over $\mathcal{ST}(\mathcal{D})$

Evaluating an assignment in the lifted domain may be performed using $F_{\mathcal{D}}$ and $F_{\mathcal{ST}}$. We construct the updated definition of x in η, then assign the corresponding 'variable' in $\mathcal{D}$ to the result of the computation.

$$\begin{aligned}
&F_{\mathcal{ST}(\mathcal{D})}[\![\mathtt{x} := \mathtt{f}(\mathtt{y_1}, \ldots, \mathtt{y_n})]\!](\langle \eta, \rho \rangle) = \langle \eta', \rho' \rangle \\
&\textbf{where } \eta' = F_{\mathcal{ST}}[\![\mathtt{x} := \mathtt{f}(\mathtt{y_1}, \ldots, \mathtt{y_n})]\!]\eta \\
&\qquad\quad \rho' = F_{\mathcal{D}}[\![\mathbf{id}(\eta'(\mathtt{x})) := \mathtt{f}(\eta(\mathtt{y_1}), \ldots, \eta(\mathtt{y_n}))]\!]\rho
\end{aligned}$$

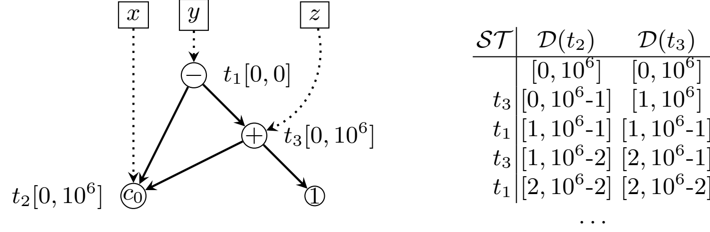
Formulating $\sqcup_{\mathcal{ST}(\mathcal{D})}$ and $\widetilde{\sqcap}_{\mathcal{ST}(\mathcal{D})}$ is only slightly more involved, assuming the presence of a *renaming* operator over $\mathcal{D}$. We first determine the term structure η' of the result, then map ρ_1 and ρ_2 onto the terms in η' before applying the join or meet.

$$\begin{aligned}
&\langle \eta_1, \rho_1 \rangle \sqcup_{\mathcal{ST}(\mathcal{D})} \langle \eta_2, \rho_2 \rangle = \langle \eta', \rho' \rangle \\
&\textbf{where } \eta' = \eta_1 \sqcup_{\mathcal{ST}} \eta_2 \\
&\qquad\quad \rho' = \pi^{\eta_1 \mapsto \eta'}(\rho_1) \sqcup_{\mathcal{D}} \pi^{\eta_2 \mapsto \eta'}(\rho_2)
\end{aligned}$$

$$\begin{aligned}
&\langle \eta_1, \rho_1 \rangle \,\widetilde{\sqcap}_{\mathcal{ST}(\mathcal{D})} \langle \eta_2, \rho_2 \rangle = \langle \eta', \rho' \rangle \\
&\textbf{where } \eta' = \eta_1 \,\widetilde{\sqcap}_{\mathcal{ST}}\, \eta_2 \\
&\qquad\quad \rho' = \pi^{\eta_1 \mapsto \eta'}(\rho_1) \sqcap_{\mathcal{D}} \pi^{\eta_2 \mapsto \eta'}(\rho_2)
\end{aligned}$$

```
x = ⋆; y = ⋆; assert(x ≥ 0)^(1)
z = x ∗ y^(2)
assert(z > 0)^(3)
```

Fig. 8: If the point (3) is reached, y must be positive.



$\mathcal{ST}$	$\mathcal{D}(t_2)$	$\mathcal{D}(t_3)$
	$[0, 10^6]$	$[0, 10^6]$
t_3	$[0, 10^6\text{-}1]$	$[1, 10^6]$
t_1	$[1, 10^6\text{-}1]$	$[1, 10^6\text{-}1]$
t_3	$[1, 10^6\text{-}2]$	$[2, 10^6\text{-}1]$
t_1	$[2, 10^6\text{-}2]$	$[2, 10^6\text{-}2]$
	...	

Fig. 9: A system of terms with no solution; encoding $x = x+1$. Each evaluation of t_1 or t_3 eliminates only two values from the corresponding bounds.

4.2 Inferring properties from subterms

While this allows us to maintain approximations of subterms, we cannot use this to directly derive tighter approximations of program variables.

However, upon encountering a branch which restricts x, we can then infer properties on any other terms involving x. For now, we shall restrict ourselves to ancestors of x. If the approximation of x has changed, and p is an immediate parent of x, we can simply *recompute* p from its definition:

$$\rho' = \rho \sqcap F_{\mathcal{ST}}[\![\mathbf{id}(\eta(\mathtt{p})) := \mathbf{def}(\eta(\mathtt{p}))]\!]\rho$$

We can then propagate this information upwards.

We can also infer information about a term from its parents and siblings. Assume the program fragment in Figure 8 is being analysed using the (term-lifted) domain of intervals. At point (1), we know only that x is non-negative; this is not sufficient to infer bounds on z. However, when point (3) is reached we know $z > 0$. As we already know $x \geq 0$, this can only occur if $y > 0$, $x > 0$.

This requires us to reason about the values from which a given computation could have resulted; this is exactly the pre-image $F^{-1}_{\mathcal{D}}$ discussed in Section 2. We can then augment the algorithm to propagate information in both directions, evaluating $F_{\mathcal{D}}$ and $F^{-1}_{\mathcal{D}}$ on each term until a fixpoint is reached. Unfortunately, attempts to fully reduce an abstract state run into difficulties.

Example 5. Consider the system of terms shown in Figure 9, augmenting the domain of intervals. Disregarding interval information, it encodes the constraint $y = x - z, z = x + 1$. In the context of $y = 0$ (the interval bounds for y), this is clearly unsatisfiable.

Propagating the consequences of these terms, we first apply the definition $t_3 = t_2 + 1$. Doing so, we trim 0 from the domain of t_3 (or z), and 10^6 from the

```
tighten(⟨η, ρ⟩, iters):
  while(iters > 0)
    ρ' := tighten-step(⟨η, ρ⟩)
    if (ρ' = ρ ∨ ρ = ⊥)
      return ρ
    ρ := ρ'
    iters := iters − 1

tighten-step(⟨η, ρ⟩):
  let t_1, ..., t_m be terms in η in
    order of decreasing height
  for t ∈ t_1, ..., t_m
    ρ := ρ ⊓ F_D^{-1}[[id(t) = def(t)]]ρ
  for t ∈ t_m, ..., t_1
    ρ := ρ ⊓ F_D[[id(t) = def(t)]]ρ
  return ρ
```

Fig. 10: Applying a system of terms η to tighten a numeric approximation ρ.

domain of t_2 (or x). We then evaluate the definition $t_1 = t_2 - t_3$, thus removing 0 and 10^6 from t_1 and t_3 respectively. We can then evaluate the definitions of t_3 and t_1 again, this time eliminating 2 and 10^6-1. This process eventually determines unsatisfiability, but it takes 10^6+1 steps to do so.[5]

This rather undermines our objective of efficiently combining $\mathcal{ST}$ with $\mathcal{D}$. If $\mathcal{D}$ is not finite, the process may not terminate at all. Consider the case where $\mathcal{D}(t_2) = \mathcal{D}(t_3) = [0, \infty]$ – the resulting iterates form an infinite descending chain, where the lower bounds are tightened by one at each iteration step.

The existence of an efficient, general algorithm for normalizing $\langle \eta, \rho \rangle$ seems doubtful. Even for the specific case of finite intervals, computing the fixpoint of such a system of constraints is NP-complete [3] (in the weak sense – the standard Kleene iteration runs in pseudo-polynomial time). Nevertheless, we can apply the system of terms to ρ some bounded number of times in an attempt to improve precision; a naive iterative approach is given in Figure 10.

In practice, this iteration is wasteful. In an independent attribute domain, applying $[\![t = \mathtt{f}(c_1, \ldots c_k)]\!]$ cannot directly affect terms not in $\{t, c_1, \ldots, c_k\}$, and we can easily detect which of these have changed. So we adopt a worklist approach, updating terms with changed abstractions only. The tightening still progresses level by level, to collect the tightest abstraction of each term before re-applying the definitions. The algorithm is outlined in Figure 11.

tighten-worklist incrementally applies a single pass of tighten-step, where only terms in X have changed. Given the discussion above, the algorithm obviously misses opportunities for propagation; this loss occurs at the point marked †. Given some definition $[\![\mathtt{t} = \mathtt{f}(\mathtt{c}_1, \mathtt{c}_2)]\!]$ and new information about c_1, we could potentially tighten the abstraction of *both* t and c_2; however, tighten-worklist only applies this information to t.

It is sound to apply the same algorithm when $\mathcal{D}$ is relational; however, it may miss further potential tightenings, as additional constraints on some term can be reflected in other, apparently unrelated terms.

[5] This behaviour is also a well recognized problem for finite domain constraint solvers (see e.g. [11]).

```
tighten-worklist(X, ⟨η, ρ⟩):
    forall l, Q↓_l := Q↑_l := ∅
    for(x ∈ X) Q↓_height(x) := Q↓_height(x) ∪ {x}
    l_min := min_{x∈X} height(x)
    l := l_max := max_{x∈X} height(x)
    while(l ≥ l_min)
        for(t ∈ Q↓_l)
            enqueue_parents(t)
            ρ' := ρ ⊓ F_D^{-1}([[id(t) = def(t)]])ρ
            for(c ∈ children(t))
                if(changed(c, ρ, ρ')) enqueue_down(c)
            ρ := ρ'
        l := l − 1
    l := l_min
    while(l ≤ l_max)
        for(t ∈ Q↑_l)
(†)         ρ' := ρ ⊓ F([[id(t) = def(t)]])ρ
            if(changed(t, ρ, ρ')) enqueue_parents(t)
            ρ := ρ'
        l := l + 1
    return ρ

enqueue_down(t):
    Q↓_height(t) := Q↓_height(t) ∪ {t}
    l_min := min(l_min, height(t))

enqueue_parents(t):
    for(p in parents(t))
        Q↑_height(p) := Q↑_height(p) ∪ {p}
        l_max := max(l_max, height(p))
```

Fig. 11: An incremental approach for applying a single iteration of tighten-step.

5 Other Syntactic Approaches

As mentioned, the closest relatives to the term domain are the symbolic constant domain of Miné [21] and the congruence closure (or *alien expression*) domain of Chang and Leino [5]. Both domains record a mapping between program variables and terms, with the objective of enriching existing numeric domains.

The term domain can be viewed as a generalization of the symbolic constant domain. Both domains arise from the observation that abstract domains, be they relational or otherwise, exhibit coarse handling of expressions outside their *native language* – particularly non-linear expressions. And both store a mapping from variables to defining expressions. The primary difference is in the join. Faced with non-equal definitions, the symbolic constant domain discards both entirely. The term domain instead attempts to preserve whatever parts of the computation are shared between the abstract states, which it can then use to improve precision in the underlying domain.

The congruence closure domain [5] arises from a different application – coordinating a heterogeneous set of base abstract domains, each supporting only a subset of expressions appearing in the program. Functions which are *alien* to a domain are replaced with a fresh variable; equivalences are inferred from the syntactic terms, and added to the base abstract domains. The congruence closure

domain assumes the base domains are relational, maintaining a system of equivalences and supported relations. As a result, it assumes the base domain will take care of maintaining relationships between interpreted expressions and the corresponding subterms. Hence it will not help with the examples from Figure 1.

While the underlying techniques are similar, the objectives (and thus trade-offs) are quite different. The congruence closure domain maintains an arbitrary (though finite) system of uninterpreted function equations, allowing multiple – possibly cyclic – definitions for subterms. This potentially preserves more equivalence information than the acyclic system of the subterm domain, but increases the cost and complexity of various operations (notably the join). As far as we know, no experimental evaluation of the congruence-closure domain has been published.

6 Experimental Evaluation

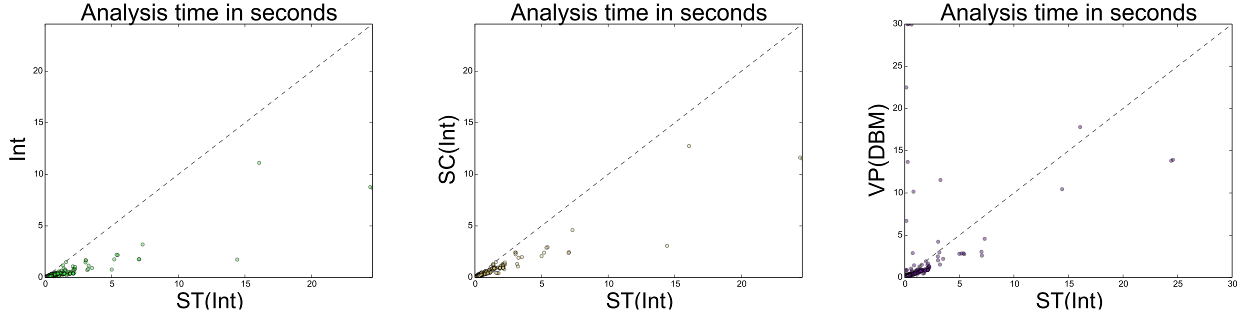
The subterm domain has been implemented in CRAB, a language-agnostic C++ library of abstract domains and fixpoint algorithms. Both the subterm domain and the rest of CRAB is available at `https://github.com/seahorn/crab`. One of the purposes of CRAB is to enhance verification tools by supplying them with inductive invariants that can be expressed in some abstract domain chosen by the client tool. For our experiments we used SeaHorn [13], one of the participants in SV-COMP 2015 [1].

We selected 2304 programs from SV-COMP 2015. We focused on the categories better supported by SeaHorn: ControlFlowInteger, Loops, Sequentialized, DeviceDrivers64, and ProductLines.[6] We first evaluated the performance of the subterm domain by measuring only the time to generate the invariants without running SeaHorn. We compared the subterm domain enhancing intervals $\mathcal{ST}(\mathsf{Intv})$ with three other numeric abstract domains: classical intervals Intv [6] which we consider the *baseline* abstract domain since it was the one used by SeaHorn in SV-COMP 2015, the symbolic constant propagation $\mathcal{SC}(\mathsf{Intv})$ [21], and an optimized implementation of difference-bound matrices using *variable packing* $\mathcal{VP}(\mathsf{DBM})$ [24]. Second, we measured the precision gains using $\mathcal{ST}(\mathsf{Intv})$ as an invariant supplier for SeaHorn and compared again with Intv, $\mathcal{SC}(\mathsf{Intv})$, and $\mathcal{VP}(\mathsf{DBM})$. All experiments were carried out on a AMD Opteron Processor 6172 with 12 cores running at 2.1GHz Core with 32GB of memory.

Performance. Table 2(a) shows three scatter plots of analysis times comparing $\mathcal{ST}(\mathsf{Intv})$ with Intv (on the left), with $\mathcal{SC}(\mathsf{Intv})$ (in the middle), and with $\mathcal{VP}(\mathsf{DBM})$ (on the right). Table 2(b) shows additional statistics about the analysis of the 2304 programs. For this experiment, we set a limit of 30 seconds and 4GB per program.

CRAB using $\mathcal{ST}(\mathsf{Intv})$, Intv, and $\mathcal{SC}(\mathsf{Intv})$ inferred invariants successfully for all programs without any timeout (column TO in Table 2(b)). The total time (denoted by $\mathsf{T}_{\mathsf{total}}$) indicates that Intv was the fastest with 175 seconds and

[6] In Table 3, we use abbreviations CFI, Loops, DD64, Seq, and PL, respectively.



(a) Scatter plots of analysis time

Domain	TO	$\mathsf{T}_{\mathsf{total}}$	T_{μ}	T_{σ}	$\mathsf{T}_{\mathsf{max}}$
Intv	0	175.4	0.08	0.38	11.12
$\mathcal{SC}$(Intv)	0	265.0	0.11	0.49	12.75
$\mathcal{ST}$(Intv)	0	456.0	0.19	0.96	24.57
$\mathcal{VP}$(DBM)	3	441.7	0.19	1.41	30.00

(b) Analysis times (seconds)

Table 2: Performance of several abstract domains on SV-COMP'15 programs

	CFI (48)		Loops (142)		DD64 (1256)		Seq (261)		PL (597)	
	#S	T	#S	T	#S	T	#S	T	#S	T
Sea+Intv	41	1589	115	5432	1215	6283	109	26031	538	20818
Sea+$\mathcal{SC}$(Intv)	41	1613	115	5480	1215	6520	110	25639	539	20741
Sea+$\mathcal{ST}$(Intv)	41	1416	121	4274	1215	6557	110	25469	542	20763
Sea+$\mathcal{VP}$(DBM)	41	1529	117	5071	1214	6854	110	25929	536	20787

Table 3: SeaHorn results on SV-COMP 2015 enhanced with abstract domains

$\mathcal{ST}$(Intv) the slowest with 456. The columns T_{μ} and T_{σ} denote the time average and standard deviation per program, and the column $\mathsf{T}_{\mathsf{max}}$ is the time of analyzing the program that took the longest. All domains displayed similar memory usage. Again, Intv was the most efficient with an average memory usage per program of 31MB and a maximum of 1.34GB whereas $\mathcal{ST}$(Intv) was the least efficient with an average of 37MB and maximum of 1.52GB.

It is not surprising that Intv and $\mathcal{SC}$(Intv) are faster than $\mathcal{ST}$(Intv); interestingly, the evaluation suggests that in practice $\mathcal{ST}$(Intv) incurs only a modest constant-factor overhead of around 2.5. $\mathcal{VP}$(DBM) was faster than $\mathcal{ST}$(Intv) in many cases but was more volatile, reaching the timeout in 3 cases. This is due to the size of variable packs inferred by $\mathcal{VP}$(DBM) [24]. If few interactions are discovered, the packs remain of constant size and the analysis collapses down to Intv. Conversely, if many variables are found to interact, the analysis degenerates into a single DBM with cubic runtime.

Precision. Table 3 shows the results obtained running SeaHorn with crab using the four abstract domains. We run SeaHorn on each verification task[7] and count the number of tasks solved (i.e., SeaHorn reports "safe" or "unsafe") shown in columns labelled with #S. In T columns we show the total time in seconds for solving all tasks. The top row gives, in parentheses, the number of programs per category. The row labelled Sea+Intv shows the number of tasks solved by SeaHorn using the interval domain (our baseline domain) as invariant supplier, while rows labelled with Sea+$\mathcal{SC}$(Intv), Sea+$\mathcal{ST}$(Intv) and Sea+$\mathcal{VP}$(DBM) are similar but using $\mathcal{SC}$(Intv), $\mathcal{ST}$(Intv) and $\mathcal{VP}$(DBM), respectively. We set resource limits of 200 seconds and 4GB for each task. In all configurations, we ran SeaHorn with Spacer [16] as back-end solver[8].

The results in Table 3 demonstrate that the subterm domain can produce significant gains in some categories (e.g., Loops and PL) and stay competitive in all. We observe that $\mathcal{SC}$(Intv) rarely improves upon the results of Sea+Intv. Two factors appear to contribute to this: the join operation on $\mathcal{SC}$(Intv) maintains only the definitions that are constant on all code paths; and SeaHorn's frontend (based on LLVM [17]) applies linear constant propagation, subsuming many of the opportunities available to $\mathcal{SC}$(Intv). Our evaluation also shows that the subterm domain helps SeaHorn solve more tasks than $\mathcal{VP}$(DBM) in several categories. One reason could be that $\mathcal{VP}$(DBM) does not perform propagation across different packs and so it is less precise than classical DBMs [19][9] and indeed incomparable with the subterm domain. Another reason might be the more precise modelling of non-linear operations by the subterm domain. Nevertheless, we observed that sometimes $\mathcal{ST}$(Intv) can solve tasks that $\mathcal{VP}$(DBM) cannot, and vice versa. For PL, for example, Sea+$\mathcal{ST}$(Intv) solved 9 tasks for which Sea+$\mathcal{VP}$(DBM) reached a timeout but Sea+$\mathcal{VP}$(DBM) solved 3 tasks that Sea+$\mathcal{ST}$(Intv) missed. This is relevant for tools such as SeaHorn since it motivates the idea of running SeaHorn with a portfolio of abstract domains.

7 Conclusion and Future Work

We have introduced the subterm abstract domain $\mathcal{ST}$, and outlined its application as a functor domain to improve precision of existing analyses. Experiments on software verification benchmarks have demonstrated that $\mathcal{ST}$, when used to enrich an interval analysis, can substantially improve generated invariants while only incurring a modest constant factor performance penalty.

The performance of $\mathcal{ST}$ is obtained by disregarding algebraic properties of operations. Extending $\mathcal{ST}$ to take exploit these properties while preserving performance poses an interesting future challenge.

[7] A program with its corresponding safety property also provided by the competition.

[8] We used the command `sea pf --step=large --track=mem` (i.e., large-block encoding [2] of the transition system modelling both pointer offsets and memory contents). For DD64 we add the option `-m64`

[9] We used an implementation of the classical DBM domain following [19] for the experiment in Table 2 but it took more than three hours to complete.

References

1. D. Beyer. Software verification and verifiable witnesses (report on SV-COMP 2015). In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015.
2. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In A. Biere and C. Pixley, editors, *Proceedings of the Ninth International Conference on Formal Methods in Computer-Aided Design*, pages 25–32. IEEE Comp. Soc., 2009.
3. L. Bordeaux, G. Katsirelos, N. Narodytska, and M. Y. Vardi. The complexity of integer bound propagation. *Jornal of Artificial Intelligence Research (JAIR)*, 40:657–676, 2011.
4. M. Bouaziz. TreeKs: A functor to make numerical abstract domains scalable. *Elctronic Notes in Theoretical Computer Science*, 287:41–52, 2012.
5. B. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2005.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, 1976.
7. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
8. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.
9. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.
10. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.
11. T. Feydy, A. Schutt, and P. Stuckey. Global difference constraint propagation for finite domain solvers. In S. Antoy, editor, *Proceedings of 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 226–235. ACM Press, 2008.
12. G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Abstract interpretation over non-lattice abstract domains. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2013.
13. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, volume 9207 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.
14. J. M. Howe and A. King. Logahedra: A new weakly relational domain. In Z. Liu and A. P. Ravn, editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science. Springer, 2009.
15. G. Huet. *Résolution d'Équations dans des Langages d'Ordre 1, 2, . . . , ω*. Thèse d'État. Université Paris VII, 1976.
16. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic abstraction in SMT-based unbounded software model checking. In N. Sharygina and H. Veith,

editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 846–862. Springer, 2013.
17. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86. IEEE Comp. Soc., 2004.
18. F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 184–188. ACM Press, 2008.
19. A. Miné. A new numerical abstract domain based on difference-bound matrices. In O. Danvy and A. Filinski, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.
20. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
21. A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In E. A. Emerson and K. S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2006.
22. A. Simon and A. King. The two variable per inequality abstract domain. *Higher-Order and Symbolic Computation*, 23(1):87–143, 2010.
23. A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2002.
24. A. Venet and G. Brat. 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*, pages 231–242. ACM Press, 2004.
25. A. J. Venet. The gauge domain: Scalable analysis of linear inequality invariants. In P. Madushan and S. A. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2012.