Lifting Numerical Abstract Domains to Heap-manipulating Programs

Zhoulai Fu, Thomas Jensen, David Pichardie

Lifting Numerical Abstract Domains to Heap-manipulating Programs

Zhoulai Fu[1], Thomas Jensen[2], and David Pichardie[3]

[1] IMDEA / IRISA - Université Rennes 1
[2] INRIA Rennes
[3] Harvard University / INRIA Rennes

Abstract. The abstract interpretation literature is rich with numerical abstract domains that allow to infer numerical properties on scalar program variables. Unfortunately, lifting this domains to heap-manipulating programs is not obvious. On the other hand, points-to analyses have been intensively studied and some scale to very large programs but without inferring any numerical properties. We propose a framework based on the theory of abstract interpretation that is able to combine existing numerical domains and points-to analyses in a modular way. Our approach allows us to prove the soundness of the combined analysis based on the soundness hypotheses of the existing analyses, and it also facilitates the modular implementation of the analysis. We have prototyped this approach using the SOOT framework for pointer analyses and the PPL library as numerical domains. Our experiments on the `DaCapo-2006` benchmark suite show that the combined analysis discovers significantly more numerical invariants than traditional numerical domains with modest time overhead.

Keywords: Abstract Interpretation, Static Numerical Analysis, Points-to Analysis

1 Introduction

The static analysis of numerical properties of program variables can draw on a rich body of techniques including abstract domains of intervals [9], polyhedron [13], octagons [25] which have found their way into mature implementations. In a similar way, the analysis of properties describing the shape of data structures in the heap has flourished into a rich set of points-to and alias analyses which also have provided a range of production-quality analyzers. However, these two types of analyses do not always integrate so well. Local numerical properties such as $x.v + y.w \leq a[i]$ (where $x.v$ and $y.w$ are Java field references of type `int` and $a[i]$ is an array reference of type `int`) are *alien* [6] to traditional numerical domains and would thus be coarsely over-approximated as *unknown*, representing no information.

When extending numerical analyses to entities such as $x.v$ we are immediately faced with the problem that pointers introduce *aliases* which make program reasoning harder. As an example, consider the effect of the assignment

$$a.val = b.val + c.val \tag{1}$$

The variables a, b, c are bound to objects with the numerical field val. Assuming that $b.val \in [3, 6]$ and $c.val \in [4, 8]$ hold before the statement, we can derive different properties for their values after the assignment depending on the knowledge we have about aliasing between the references a, b and c.

In particular, the values of $b.val$ or $c.val$ may be updated if the condition $a = b$ or $a = c$ holds before the statement. The following approach considers the potential aliases among variables a, b and c. There are five possible alias relations, as shown on columns 2-6 of the first row in Tab. 1, where we use '/' to mean the partitions of variables induced by aliasing. For example, in the case of ab/c, the alias relation is $a = b \neq c$, and thus $a.val$, $b.val$ are two names that must be updated simultaneously. We obtain $a.val \in [7, 14], b.val \in [7, 14]$ and $c.val \in [4, 8]$. The last column of the table shows the post-conditions by joining the results in columns 2-6.

Table 1: Post-conditions of $a.val = b.val + c.val$, assuming $b.val \in [3, 6], c.val \in [4, 8]$. Columns 2-6 show 5 aliasing relations between 3 variables. The last column joins the results.

	$a/b/c$	ab/c	ac/b	bc/a	abc	join
a.val	[7,14]	[7,14]	[7,14]	[8,12]	[8,12]	[7,14]
b.val	[3,6]	[7,14]	[3,6]	[4,6]	[8,12]	[3,14]
c.val	[4,8]	[4,8]	[7,14]	[4,6]	[8,12]	[4,14]

Analyzing the statement for every possible alias relation between variables in turn and taking the conservative join of obtained results gives a sound result. However, this naive approach is not feasible as the number of aliasing relations among N variables quickly becomes large[1]. A better solution is to combine traditional static numerical analysis with points-to analyses that can provide information about aliasing relations and hence rule out some spurious aliases.

1.1 Objectives and Contributions

This paper is concerned with developing a theoretical foundation for combining pointer analysis with static numerical analysis. The goal is not to define new pointer and numerical analyses but to provide the necessary theory for interfacing existing analyses with each other. We shall be following the methodology of

[1] The number of aliasing relations is the number of partitions of a n-element set (known as the *Bell number*) and is asymptotically $O(\epsilon^n n!)$ for any positive ϵ [5]

abstract interpretation [12] when constructing the theory. The contributions of the paper are both theoretical and practical. On the theoretical side, we propose a new abstract domain combining traditional static numerical domains and points-to analysis. The abstract domain is constructed in three steps:

1. the first is a lattice isomorphism in which the references in the heap part of the state are re-injected into (and hence made explicit in) the numerical part of the state,
2. the second is a Cartesian (attribute-independent) abstraction [12] of the numerical and the heap part of the state,
3. the third is the application of the abstractions of the existing domains.

Thus, it is the first step that makes the combination possible, by preparing the re-use of the abstract pointer values when extending the numerical domains to cover properties about heap values. We define and prove the correctness of the transfer functions for this new combined domain.

On the practical side, we have experimented with the combination of several existing domains by implementing a combined static numerical and pointer analysis, using the Java Optimization Framework SOOT [34] as the front-end, and relying on the abstract domains from existing static analysis libraries such as the Parma Polyhedra Library PPL [2] and the Soot Pointer Analysis Research Kit SPARK [20]. This prototype analyzer, called NumP, has been run on programs in the `Dacapo-2006-MR2` [3] benchmark suite. The largest among them, *chart*, has more than 350K LOC in Jimple [35]. Our experiments confirm that a combined analysis is feasible even for large-sized programs and that it discovers significantly more program properties than what is possible by pure numerical analysis, and this at a cost that is comparable to the cost of running the numerical and pointer analysis separately.

In addition, the goal of modular re-use of static analyses has been attained as the implementation of our prototype is mainly based on the existing implementations of traditional numerical and pointer analyses. We have instanced NumP with a context-insensitive and a context sensitive points-to analyses on one side, and an interval and a polyhedral abstract domains on the other side.

1.2 Organization of the paper

The interfaces of traditional numerical and pointer analyses, as well as the target languages are specified in Sect. 2. The intuition of our analysis is illustrated with a small example in Sect. 3. In Sect. 4 we define the semantics abstraction process in three well-defined steps. The algorithm is presented in Sect. 5 with a proof sketch. Experimental results are given in Sect. 6. Finally, we compare our analysis with related work and conclude in Sect. 7 and 8. The full proof of the soundness of the abstract transfer functions is given in Appendix.

Notation Let A, B be two sets. Given a relation $R \subseteq A \times B$, we write $\mathsf{post}[R] \in \wp(A) \to \wp(B)$ for the function $\lambda A_1.\{b \mid \exists a \in A_1 : (a, b) \in R\}$. We use fst and

snd as the operators that extract the first and the second components of a pair respectively. For a given set U, the notation $U_\perp$ means the disjoint union $U \cup \{\perp\}$. Given a mapping $m \in A \to B_\perp$, we express the fact that m is undefined in a point x by $m(x) = \perp$. The set of integers is denoted by $\mathbb{Z}$. We write "$\triangleq$" for "defined as".

2 Analysis Interfaces

The purpose of this section is to define the interfaces of the two existing analyses that will be used in the paper.

2.1 The languages WHILE_{np}, WHILE_n and WHILE_p

Consider the minimal imperative language that mixes pointer and numerical operations, denoted by WHILE_{np}. Fig. 1 gives its formal syntax (left) and an example program (right), where k is a constant, $\diamond$ is an arithmetic operator and $\bowtie$ is a comparison operator. We assume two sets of variables: numerical variables and pointer variables. Variables x, y, z and field f are subscripted with n or p to indicate whether they are numerical values or pointers.

$$
\begin{aligned}
s_n &::= x_n = k \mid x_n = y_n \\
&\mid x_n = y_n \diamond z_n \mid x_n \bowtie y_n \\
s_p &::= x_p = \texttt{new} \mid x_p = x_p.f_p \\
&\mid x_p = y_p \mid x_p.f_p = y_p \\
s_{np} &::= x_p.f_n = y_n \mid x_n = y_p.f_n \\
s &::= s_n \mid s_p \mid s_{np}
\end{aligned}
$$

```
1 int i = -5;
2 A hd = null, elem = null;
3 while (i < 3) {
4   elem = new Node(); //h
5   elem.val = i;
6   elem.next = hd;
7   hd = elem;
8   i = i + 1;
9 }
```

Fig. 1: The basic statements of WHILE_{np} (left) ranged over by s, and an example program (right).

We shall use WHILE_n to refer to basic statements only involving numerical variables and use the meta-variables s_n to range over those statements. Similarly, we let WHILE_p be the statements that only use pointer variables and let s_p range over those statement. Thus, the basic statements of WHILE_{np} include those in WHILE_n and WHILE_p, and two more statements in the forms of $x_p.f_n = y_n$ and $x_n = y_p.f_n$, where x_p, y_p are pointer variables, x_n, y_n are numerical variables and f_n is a numerical field. We let s_{np} range over the two extra assignments statements belonging to WHILE_{np}. Meta-variable s ranges over all the statements of WHILE_{np}. Note that s_n, s_p and s_{np} are the basic statements, rather than a statement with a whole-structure.

Traditional numerical analyzers Num and traditional pointer analyzers Pter deal with s_n and s_p respectively. Our objective is to design an analysis that is able to deal with s_n, s_p, and s_{np}. We shall assume that Num and Pter are available, and consider the problem of how to systematically combine them in a "black-box" manner so that an abstract semantics for s can be derived while maintaining the precisions of s_n and s_p. This combined analyzer will be denoted as NumP.

2.2 Analysis of WHILE_n

We use the term *numerical property*, for any conjunction of formulae in some decidable theory of arithmetic. A numerical property can be loosely seen as a geometric shape. For example, the numerical property $\{x^2+y^2 \leq 1, \quad x \leq 0, \quad y \leq 0\}$ is composed of the conjunction of three arithmetic formula, representing a quart of the unit disc. Each formula of a numerical property is assumed to be quantifier-free. The constant values in the formula are integers.

As usual, an environment is a partial mapping from program variables to their associated values. In our context, we consider *numerical environment* of integer values. $Num \triangleq Var_n \to \mathbb{Z}_\perp$ where Var_n is the set of numerical variables.

The relationship between a numerical environment n and a numerical property $\bar{\mathsf{n}}$ is formalized by the concept of *valuation*. We say that n is a valuation of $\bar{\mathsf{n}}$, denoted by $\mathsf{n} \models \bar{\mathsf{n}}$, if $\bar{\mathsf{n}}$ becomes a tautology after each of its free variables, if any, has been replaced by its corresponding value in n.

Definition 1 (Interface of the traditional numerical analyzer).

$$(\mathsf{WHILE}_n, \wp(Num), [\![\cdot]\!]_n^\natural, \gamma_n, Num^\sharp, [\![\cdot]\!]_n^\sharp)$$

The concrete numerical domain *and the* abstract numerical domain *for the language* WHILE_n *are respectively* $\wp(Num)$ *and* $Num^\sharp$. *They are related by the* concretization function $\gamma_n : Num^\sharp \to \wp(Num)$ *defined by*

$$\gamma_n(\bar{\mathsf{n}}) = \{n \in Num \mid n \models \bar{\mathsf{n}}\} \tag{2}$$

The partial order $\sqsubseteq$ *is consistent with the monotonicity of* γ_n, i.e., $\bar{\mathsf{n}}_1 \sqsubseteq \bar{\mathsf{n}}_2$ *implies* $\gamma_n(\bar{\mathsf{n}}_1) \subseteq \gamma_n(\bar{\mathsf{n}}_2)$. *For each statement* s_n *of* WHILE_n, *the concrete semantics* $[\![s_n]\!]_n^\natural$ *is assumed to be the powerset lifting* $[\![s_n]\!]_n^\natural \triangleq \mathsf{post}[\xrightarrow{Num}(s_n)]$ *of some standard operational semantics:*

$$\xrightarrow{Num}: \mathsf{WHILE}_n \to \wp(Num \times Num) \tag{3}$$

The abstract semantics $[\![\cdot]\!]_n^\sharp$ *satisfies the soundness condition:*

$$[\![\cdot]\!]_n^\natural \circ \gamma_n \subseteq \gamma_n \circ [\![\cdot]\!]_n^\sharp \tag{4}$$

At last, we assume the availability of a join operator $\sqcup$ *and a widening operator* $\bigtriangledown$. *The join operator is assumed to be sound with regard to the partial order* $\sqsubseteq$, *and* $\bigtriangledown$ *is assumed to be sound as specified in Sect. 4 of [11].*

2.3 Analysis of WHILE_p

The imperative language WHILE_p provides basic pointer operations like dynamic allocation, pointer assignments, field store and field load.

A state in WHILE_p is typically thought of as a graph-like structure representing the *environment* and *heap*. Let *Ref* denote the memories involving in program run-time. Similar to the notation used in [27], we use the triple (A, ρ, hp) to denote a memory state, where A keeps track of the allocated memory locations, ρ is a partial mapping from variables to references, and hp is a partial mapping from fields and references to references. The set of the triples is denoted by *Pter*.

$$Pter \triangleq \wp(Ref) \times (Var_p \to Ref_\perp) \times ((Ref \times Fld_p) \to Ref_\perp) \tag{5}$$

The points-to analysis [15] is a dataflow analysis widely used for the static pointer analysis. The essential idea of points-to analysis is to partition *Ref* into a finite set H, and then summarize the run-time pointer relations via elements of H and program variables. The result of the analysis is often expressed by a graph-like structure, called *points-to graph*. The memory partition process mentioned above is sometimes called a *naming scheme*. A popular naming scheme, known as k-CFA [29], is based on the k most recent call sites on the stack of the thread creating the object. We assume that a *naming scheme* can be interfaced with a function

$$\rhd \in Ref \to H \tag{6}$$

In this presentation, we use a simple and standard naming scheme to name heap elements after the program point of the statement that allocates them (which is typical for the *context-insensitive* variant of points-to analysis). The elements of H will also be called *allocation sites* or *abstract references*.

Definition 2 (Interface of traditional pointer analyzer).

$$(\mathsf{WHILE}_p, \wp(Pter), [\![\cdot]\!]_p^\natural, \gamma_p, Pter^\sharp, [\![\cdot]\!]_p^\sharp)$$

The concrete heap domain *and the* abstract heap domain *for the language* WHILE_p *are respectively* $\wp(Pter)$ *and* $Pter^\sharp$. *They are related by some monotone* concretization function $\gamma_p : Pter^\sharp \to \wp(Pter)$. *The concrete semantics* $[\![\cdot]\!]_p^\natural$ *is assumed to be the powerset lifting of some standard operational semantics* $\xrightarrow{Pter}$: $\mathsf{WHILE}_p \to \wp(Pter \times Pter)$. *The soundness is assumed* $[\![\cdot]\!]_p^\natural \circ \gamma_p \subseteq \gamma_p \circ [\![\cdot]\!]_p^\sharp$.

3 Combining Points-to and Numerical Analysis: Intuition

In this section we will present the intuition behind the technique for combining points-to analyses and numerical analyses. The idea is to use the names computed by a points-to analysis to create *symbolic variables* that can represent the numerical values stored at particular heap locations. The formal underpinning and semantic correctness of the combination technique are presented in Sect. 4 and 5.

As an illustrative example, consider the Java snippet in Listing. 1.1. In this example, an abstract class `Unsigned` uses unsigned numbers to represent both positive and negative values. `Unsigned` has two subclasses `Pos` and `Neg` for this purpose. It is the responsibility of clients to ensure the underlined contract, *i.e.*, objects of type `Unsigned` must hold non-negative values. The Java source code takes an array *buf* and passes the elements to the list *elem* of type `List`. The `List` has a field *item* for data type `Unsigned` and a field *next* of type `List`. The compound condition structure (l. 7-14 of Listing. 1.1) creates an object of `Pos` or `Neg` according to whether n is positive or not. For both cases, *data.val* will be assigned to the absolute values of n so that the assumed property of unsignedness can be preserved. From l. 15 to l. 19, the program allocates a new cell to store *data* and link it to the list created from the precedent iteration.

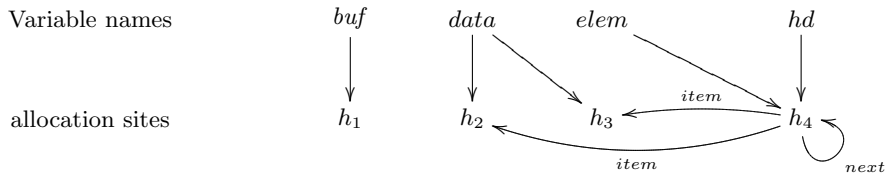
Our analysis is able to infer the following properties at the end of the program (l .21).

- **Prop1** Each list element of *hd* is in the range of 0 to 9:

$$\forall l \geq 0, hd.next^{l}.item.val \in [0.9]$$

- **Prop2** Each array element of *buf* is in the range of -9 to 7: $buf[*] \in [-9, 7]$
- **Prop3** The loop index *idx* is equal to or larger than the length of the array buf: $idx \geq buf.length$.

We start with a flow-insensitive points-to analysis. A single points-to graph for the whole program can be obtained. The graph has two kinds of arcs. Unlabelled arcs $v \rightarrow h$ from a variable v to an allocation site h, and labeled arcs $h \xrightarrow{f} h'$ between allocation sites h, h' with field f as label.



Semantically, the points-to graph disambiguates the heap and tells what must not alias. In line with this semantics, we derive a symbolic variable $\delta_{(h,val)}$ for each pair of heap location h and field *val*. The key insight is, numerical values bound to syntactically distinct symbolic variables are guaranteed to be stored at different concrete heap locations. It is therefore reasonable to deal with symbolic variables like with scalar variables.

We associate $buf[i]$ at l. 1 of Listing. 1.1 with a symbolic variable δ_1, and $buf.length$ at l. 5 with δ_0. Because variable *data* point to h_2 and h_3, we associate *data.val* at l. 9 and l. 13 with a compound symbolic variables $\delta_{2,3}$, reflecting the fact that *data* may be bound to an object `Pos` or `Neg`. Listing 1.2 illustrates the semantics actions taken by our analysis. It is worth noting that more than one run-time heap locations may be associated with the same symbolic variable,

e.g., δ_1 corresponds to all heap locations of the array *buf*. By updating the symbolic variable to -9, 7, 3 and -5 successively, we perform a *weak update*, *i.e.*, *accumulating* values rather than *overwriting* them. A formal account on the symbolic variable will be shown in the next two sections.

To sum up, a heap cell allocated by instruction at allocation site h and containing a field *val* would give rise to a symbolic variable δ. Then we extend the set of numerical variables with these new symbolic variables and perform a numerical analysis (interval, polyhedral,...) on this extended set of variables. Finally, the analysis of the program in Listing 1.1 can be treated as an *extended* numerical analysis, with its semantics actions specified in Listing 1.2. This analysis is called "extended" because it not only deals with scalar variables, but also deals with symbolic variables. By performing an extended polyhedral analysis, we are able to infer the four invariants at l. 21: $\delta_{2,3} \in [0, 9]$, $\delta_1 \in [-9, 7]$ and $\delta_0 - idx \leq 0$, which imply **Prop1**, **Prop2** and **Prop3** respectively.

```
1  int[] buf = {-9,7,3,-5};//h1
2  Unsigned data = null;
3  List hd = null;
4  int idx = 0;
5  while (idx < buf.length){
6   int n = buf[idx];
7   if (n > 0){
8    data = new Pos();//h2
9    data.val = n;
10  }
11  else{
12   data = new Neg();//h3
13   data.val = -n;
14  }
15  List elem = new List();//h4
16  elem.item = data;
17  elem.next = hd;
18  hd = elem;
19  idx = idx + 1;
20 }
21 return;
```

Listing 1.1: A Java snippet

```
1  δ1 = -9;
2  δ1 = 7;
3  δ1 = 3;
4  δ1 = -5;
5  idx = 0;
6  len = δ0;
7  while (idx < len){
8    n = δ1;
9    if (n > 0)
10     δ2,3 = n;
11   else
12     δ2,3 = -n;
13   idx = idx + 1;
14 }
```

Listing 1.2: Semantics actions

Fig. 2: An example in Java. The program passes an array of integers to a list of `Unsigned` numbers. `Unsigned` is a superclass of `Pos` and `Neg`. It has one field `val` of integer type. The class `List` has two fields, `item` of type `Unsigned`, and `next` of type `List`.

4 Semantics Abstraction

The store-based semantics for heap reasoning is standard. We follow the notations of [27], in which a state keeps track of the allocated references $A \in \wp(Ref)$, and a pair of an environment ρ and a heap hp.

$$(A, \rho, hp) \in State = \wp(Ref) \times \overbrace{(Var_n \to \mathbb{Z}_\perp) \times (Var_p \to Ref_\perp)}^{Env} \\ \times \underbrace{((Ref \times Fld_n) \to \mathbb{Z}_\perp) \times ((Ref \times Fld_p) \to Ref_\perp)}_{Heap}$$

Write $A^\flat$ for the powerset of $State$. In the context of WHILE_{np}, the variables and fields are typed. Let $\longrightarrow^\flat$: $S \to \wp(State \times State)$ denote its *structural operational semantics* (omitted).

4.1 An Isomorphic Operational Semantics

The lemma below shows that we can express the structural operational semantics (SOS for short) of WHILE_{np} in terms of the SOSs of WHILE_n and WHILE_p.

Lemma 1 (Isomorphic store-based semantics of WHILE_{np}). *Let Num and Pter be the sets of concrete states of WHILE_n and WHILE_p, as in Sect. 2. Let D denote the set of the pairs of concrete references in Ref and the numerical fields in Fld_n.*

$$D \triangleq Ref \times Fld_n \tag{7}$$

Let $Num[D \cup Var_n]$ be the set that extends Num over $D \cup Var$, i.e., *$Num[D \cup Var_n] \triangleq (D \cup Var_n) \to \mathbb{Z}_\perp$. Then a concrete state of WHILE_{np} is also an element in $\widetilde{State}$, with*

$$\widetilde{State} \triangleq Num[D \cup Var_n] \times Pter \tag{8}$$

In addition, The SOS of WHILE_{np}, denoted by $\widetilde{\longrightarrow}^\flat$, can be expressed by the SOSs of WHILE_n and WHILE_p. (Fig. 3, in which $\xrightarrow{Pter}$ is the SOS of WHILE_p, and $\xrightarrow{Num}$ is the SOS of WHILE_n over $D \cup Var_n$).

Let $\widetilde{A^\flat}$ be the power-set of $\widetilde{State}$, we define the *collecting semantics* [10] as the lifting of the operational semantics $\widetilde{\longrightarrow}^\flat(s)$ to power-sets, *i.e.*, $[\![s]\!]^{\tilde{\flat}} \triangleq \mathsf{post}[\widetilde{\longrightarrow}^\flat(s)]$.

4.2 Cartesian Abstraction

Lemma 2 (Cartesian Abstraction). *Let $A^\natural \triangleq \wp(Num[D \cup Var_n]) \times \wp(Pter)$ and $(\widetilde{A^\flat}, \alpha^\times, \gamma^\times, A^\natural)$ be the Cartesian abstraction [12],* i.e., *$\alpha^\times \triangleq \lambda R : \widetilde{A^\flat}.$ $(\mathsf{post}[\mathsf{fst}]\ R, \mathsf{post}[\mathsf{snd}]\ R)$ and $\gamma^\times \triangleq \lambda(A_0, B_0) : A^\natural.A_0 \times B_0$.*

The transfer functions $[\![\cdot]\!]^\natural : \mathsf{WHILE}_{np} \to (A^\natural \to A^\natural)$ defined in Fig. 4 is the best transformer *[12] of $[\![\cdot]\!]^{\tilde{\flat}}$, that is, $\forall s \in \mathsf{WHILE}_{np} : [\![s]\!]^\natural = \alpha^\times \circ [\![s]\!]^{\tilde{\flat}} \circ \gamma^\times$.*

$$\frac{\langle s_n, \mathsf{n}\rangle \xrightarrow{Num} \mathsf{n}'}{\langle s_n, (\mathsf{n},\mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}', \mathsf{p})} \qquad \frac{\mathsf{p}(y_p) = r \quad d = (r, f_n) \quad \langle d = x_n, \mathsf{n}\rangle \xrightarrow{Num} \mathsf{n}'}{\langle y_p.f_n = x_n, (\mathsf{n},\mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}', \mathsf{p})}$$

$$\frac{\langle s_p, \mathsf{p}\rangle \xrightarrow{Pter} \mathsf{p}'}{\langle s_p, (\mathsf{n},\mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}, \mathsf{p}')} \qquad \frac{\mathsf{p}(y_p) = r \quad d = (r, f_n) \quad \langle x_n = d, \mathsf{n}\rangle \xrightarrow{Num} \mathsf{n}'}{\langle x_n = y_p.f_n, (\mathsf{n},\mathsf{p})\rangle \widetilde{\longrightarrow}^{\flat} (\mathsf{n}', \mathsf{p})}$$

Fig. 3: Structural Operational semantics $\widetilde{\longrightarrow}^{\flat} : \mathsf{WHILE}_{np} \to (\widetilde{State} \times \widetilde{State})$

$$[\![s_p]\!]^{\natural}(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq (\tilde{\mathsf{n}}, [\![s_p]\!]^{\natural}_p(\tilde{\mathsf{p}})) \qquad [\![s_n]\!]^{\natural}(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq ([\![s_n]\!]^{\natural}_n(\tilde{\mathsf{n}}), \tilde{\mathsf{p}})$$

$$[\![x_n = y_p.f_n]\!]^{\natural}(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq \left(\left(\bigcup_{\tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d} [\![x_n = d]\!]^{\natural}_n(\tilde{\mathsf{n}})\right), \tilde{\mathsf{p}}\right)$$

$$[\![y_p.f_n = x_n]\!]^{\natural}(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \triangleq \left(\left(\bigcup_{\tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d} [\![d = x_n]\!]^{\natural}_n(\tilde{\mathsf{n}})\right), \tilde{\mathsf{p}}\right)$$

Fig. 4: $[\![\cdot]\!]^{\natural} : \mathsf{WHILE}_{np} \to (A^{\natural} \to A^{\natural})$. *The notation* $\tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d$ *means that* $d \in \{(\mathsf{p}(y_p), f_n) \mid \mathsf{p} \in \tilde{\mathsf{p}}\}$

Proof (Proof of Lem. 2). , for the case of $y_p.f_n = x_n$]

$$\begin{aligned}
&\alpha^{\times} \circ [\![y_p.f_n = x_n]\!]^{\tilde{\flat}} \circ \gamma^{\times}(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) \\
=&\alpha^{\times} \circ [\![y_p.f_n = x_n]\!]^{\tilde{\flat}}(\tilde{\mathsf{n}} \times \tilde{\mathsf{p}}) && \text{(Def. } \gamma^{\times}\text{)} \\
=&\alpha^{\times}(\{\widetilde{\longrightarrow}^{\flat}(y_p.f_n = x_n)(\mathsf{n},\mathsf{p}) \mid (\mathsf{n},\mathsf{p}) \in \tilde{\mathsf{n}} \times \tilde{\mathsf{p}}\}) && \text{(Def. } [\![\cdot]\!]^{\tilde{\flat}}\text{)} \\
=&\alpha^{\times}(\{(\xrightarrow{Num}(d = x_n)(\mathsf{n}),\mathsf{p}) \mid (\mathsf{n},\mathsf{p}) \in \tilde{\mathsf{n}} \times \tilde{\mathsf{p}}, d = (\mathsf{p}(y_p), f_n)\}) && \text{(Def. } \widetilde{\longrightarrow}^{\flat}\text{)} \\
=&(\{\xrightarrow{Num}(d = x_n)(\mathsf{n}) \mid (\mathsf{n},\mathsf{p}) \in \tilde{\mathsf{n}} \times \tilde{\mathsf{p}}, \tilde{\mathsf{p}} \vdash y_p.f_n \Downarrow d\}, \tilde{\mathsf{p}}) && \text{(Def. } \alpha^{\times}\text{)} \\
=&[\![y_p.f_n = x_n]\!]^{\natural}(\tilde{\mathsf{n}}, \tilde{\mathsf{p}}) && \text{(Def } [\![\cdot]\!]^{\natural}\text{)}
\end{aligned}$$

4.3 The Abstract Domain *NumP*

Definition 3 (Symbolic variable). *A symbolic variable* δ *is a pair of abstract reference* $h \in H$ *and numeric field* $f_n \in Fld_n$. *The set of symbolic variables is denoted by* Δ.

$$\Delta \triangleq H \times Fld_n \tag{9}$$

Remark that the set Δ is fixed whenever the naming schema $\triangleright$ (see Def. 6)is chosen. The role of *symbolic variables* is formalized via the notion of *instantiation.*

Definition 4 (Instantiation).

$$\mathtt{Ins}_{\triangleright} \triangleq \{\sigma : \Delta \to D \mid \sigma(h, f_n) = (r, g_n) \Rightarrow h = \triangleright(r) \wedge f_n = g_n\} \tag{10}$$

Notation Let $Num^\sharp[\Delta \cup Var_n]$ and $Num^\sharp[D \cup Var_n]$ denote the extensions of $Num^\sharp$ over $\Delta \cup Var_n$ and $D \cup Var_n$ respectively. Given $\sigma \in \mathtt{Ins}_{\triangleright}$, we denote by $[\sigma]$ the capture-avoiding substitution operator of type $Num^\sharp[\Delta \cup Var_n] \to Num^\sharp[D \cup Var_n]$ that replaces all the free occurrences of δ in $\bar{\mathsf{n}} \in Num^\sharp[\Delta \cup Var_n]$ with $\sigma(\delta)$.

Definition 5. *The semantics of elements in $Num^\sharp[\Delta \cup Var_n]$ is defined by the concretization function $\gamma_\delta : Num^\sharp[\Delta \cup Var_n] \to \wp(Num[D \cup Var_n])$*

$$\gamma_\delta(\bar{\mathsf{n}}) \triangleq \{\mathsf{n} \in Num[D \cup Var_n] \mid \forall \sigma \in \mathtt{Ins}_{\triangleright}, \mathsf{n} \in \gamma_n \circ [\sigma](\bar{\mathsf{n}})\}$$

Read it as follows: a numerical environment n over $D \cup Var_n$ is in the concretization of the numerical property $\bar{\mathsf{n}}$ over $\Delta \cup Var_n$ if and only if n is in the concretization of each instance of $\bar{\mathsf{n}}$.

Example 1. Consider $\bar{\mathsf{n}} \in Num^\sharp[\Delta \cup Var_n]$ to be a system of linear inequalities $AX \leq B$ with A and B being numerical matrix and the vector respectively, and X is a vector on $\Delta \cup Var_n$. Without loss of generality, we write X as the vector $(\delta_1 \dots \delta_m, z_1 \dots z_l)$ for $\delta_i \in \Delta$ and $z_j \in Var_n$. Then $AX \leq B$ represents the conjunction of all $A\bar{X} \leq B$ in which $\bar{X}$ can be any $(d_1 \dots d_m, z_1 \dots z_l)$ in which z_i remains the same as in X and there exists an instantiation $\sigma \in \mathtt{Ins}_{\triangleright}$ s.t. $\sigma(\delta_i) = d_i$ for any $1 \leq i \leq m$.

Definition 6 (The abstract domain *NumP* and its concretization). *The abstract domain NumP is defined to be*

$$NumP \triangleq Num^\sharp[\Delta \cup Var_n] \times Pter^\sharp \tag{11}$$

The concretization function γ'' of type $NumP \to A^\natural$ is defined as $\lambda(\bar{\mathsf{n}}, \hat{\mathsf{p}}).(\gamma_\delta(\bar{\mathsf{n}}), \gamma_p(\hat{\mathsf{p}}))$ where γ_p is the concretization function of the underlying points-to anlaysis as specified in Sect. 2.3.

Example 2. Revisit the program in Fig. 1 (right). A list of integers ranging from -5 to 2 is stored iteratively on the heap. At each iteration, a memory cell, bound to variable *elem*, is allocated. The cell consists of a numerical field *val* and a reference field *next*. The head of the list is always pointed to by variable *hd*.

Fig. 5 shows the memory states that arise at the loop entry (l. 3 of the source code) as well as the process of the semantics abstraction in three steps. The first row illustrates the concrete heap states. The second row is an isomorphic version that separates numerical and pointer information. The third row is the abstract state obtained by performing Cartesian abstraction over the second row. The last row shows the abstract state of our abstract domain. Note that each state, $(\mathsf{n}_k, \mathsf{p}_k)$ of the second row is a concretization of the abstract state $(\bar{\mathsf{n}}, \hat{\mathsf{p}})$.

In particular, the $(h, val) \to [-5, 2]$ part is to be interpreted as: the numerical values stored at (r, val) must be in the range from -5 to 2, whenever the memory cell referred by r is allocated at h.

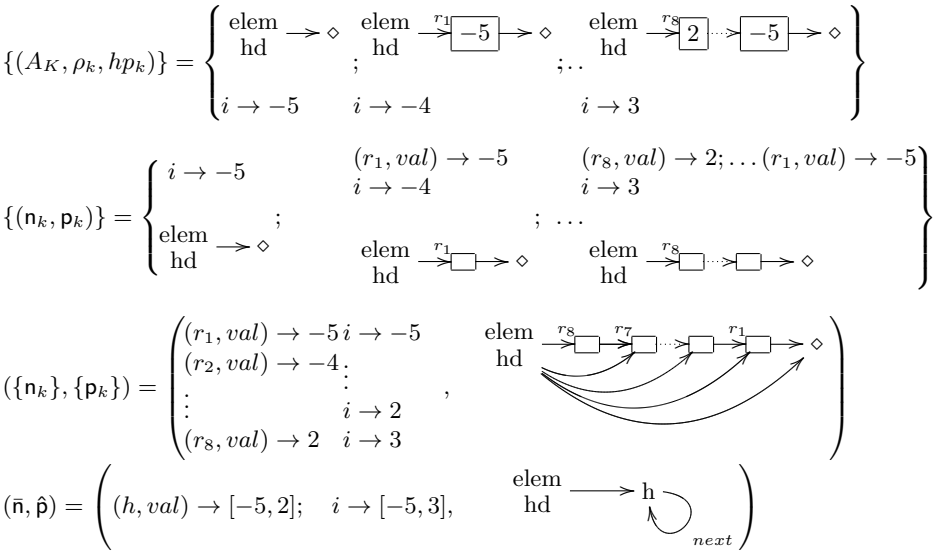


Fig. 5: Semantics abstraction of memory states at the loop entry of the example program in Fig. 1 (right, l. 3). Heap locations are depicted as rectangles labeled by references. The value of each pointer variable is depicted as an arrow from the variable name to the referenced rectangle. The symbol $\diamond$ is for the `null` pointer. We have omitted the range $1 \leq k \leq 8$ of the script k occurring in the first three rows. The label for the field "next" on the directed edges is not drawn for the first three rows.

5 Transfer functions

Let $(\bar{\mathsf{n}}, \hat{\mathsf{p}})$ be a state of *NumP*. We are concerned with how it should be updated by statements of WHILE_{np}. Let $[\![s]\!]^\sharp(\bar{\mathsf{n}}, \hat{\mathsf{p}})$ be the state just after the execution of some statement s. Below, we explain how their abstract semantics should be defined following the three kinds s_n, s_p, and s_{np} classified in Sect. 2. Note that the points-to component of our abstraction is described in a flow-sensitive style but is relatively easy to adapt it for a flow-insensitive points-to analysis. The proof of soundness is sketched at the end of the section.

Transfer function for s_n It is sound to assume that assignments or assertions of numerical variables have no effect on the heap. If s_n is an assignment in

WHILE_n, it can be treated in the same way as in traditional numerical analysis using its abstract transfer function $[\![\cdot]\!]_n^\sharp$ (as specified in Sect. 2.2).

The transfer function for updating $(\bar{\mathsf{n}}, \hat{\mathsf{p}})$ with s_n can be defined as:

$$[\![s_n]\!]^\sharp (\bar{\mathsf{n}}, \hat{\mathsf{p}}) \triangleq [\![s_n]\!]_n^\sharp \, \bar{\mathsf{n}}, \hat{\mathsf{p}} \tag{12}$$

If s_n is an assertion in WHILE_n, $\hat{\mathsf{p}}$ may be refined. For example, consider the *compound statement*[2] `if (a > 0) p = q` where p and q are reference variables and a is a numerical variable. Although it should be possible to perform a dead-code elimination using inferred numerical relations, similar to Pioli's conditional constant propagation [26], we still use the Eq. (12) for the ease of implementation.

Transfer function for s_p It is also sound to assume that s_p has no effect upon $\bar{\mathsf{n}}$. Yet the reasoning is different from the above case. For example, if $\bar{\mathsf{n}}, \hat{\mathsf{p}}$ is the state shown on the last row of Fig. 5, how can we tell whether an assignment of pointers modifies $\bar{\mathsf{n}}$ or not? Recall that the intended semantics of $(h, val) \to [-5, 2]$ is that every values stored in each (r, val) satisfying $r \rhd h$ must be in the range of $[-5, 2]$. That is to say, $\bar{\mathsf{n}}$ represents a fact about the *numerical content* stored in the corresponded concrete references. A pointer assignment can by no means modify any numerical values stored in the heap. The algorithm to update $(\bar{\mathsf{n}}, \hat{\mathsf{p}})$ with s_p can be written as:

$$[\![s_p]\!]^\sharp (\bar{\mathsf{n}}, \hat{\mathsf{p}}) \triangleq \bar{\mathsf{n}}, [\![s_p]\!]_p^\sharp \, \hat{\mathsf{p}} \tag{13}$$

Transfer function of s_{np} The transfer function for s_{np} is more interesting. Consider an assignment $x_n = y_p.f_n$. Assume that the state before the assignment is $(\bar{\mathsf{n}}, \hat{\mathsf{p}})$ with $\bar{\mathsf{n}} = \{(h1, val) \to [0, 5], (h_2, val) \to [10, 20]\}$ and $\hat{\mathsf{p}} = \{(y_p, h_1), (y_p, h_2)\}$. Since y_p points to h_1, h_2 and thus $y_p.f_n$ is bound with a subset of values stored at (r, f_n) so that $r \rhd h_1$ or $r \rhd h_2$, we know that at runtime the assignment updates x_n to a value that is either in $[0, 5]$ or in $[10, 20]$. In general, the semantics of $x_n = y_p.f_n$ can be approximated by the join of the effects of the assignment of symbolic variables, $x_n = (h, f_n)$, for all h such that y_p points to h.

$$[\![x_n = y_p.f_n]\!]^\sharp (\bar{\mathsf{n}}, \hat{\mathsf{p}}) \triangleq \left(\left(\bigsqcup_{\hat{\mathsf{p}} \vdash y_p.f_n \Downarrow \delta} [\![x_n = \delta]\!]_n^\sharp (\bar{\mathsf{n}}) \right), \hat{\mathsf{p}} \right) \tag{14}$$

where we write $\hat{\mathsf{p}} \vdash y_p.f_n \Downarrow \delta$ to mean that δ is some symbolic variable (h, f_n) with h pointed to by x_p. *i.e.*, $\exists h : \delta = (h, f_n) \wedge h \in \hat{\mathsf{p}}(x_p)$.

Consider an assignment $y_p.f_n = x_n$ with y_p pointing to the abstract h of the points-to graph. We regard $y_p.f_n = x_n$ as an assignment to symbolic variable

[2] This term is used here to be distinguished from basic statements as s_n, s_p or s_{np}. Note that s_n is the assertion, not the whole if-statement.

$(h, f_n) = x_n$. By $(h, f_n) = x_n$, we actually mean that the field f_n of *one of the concrete objects represented by* h is to be updated to the value of x_n, while the other concrete objects represented by h remains unchanged. In practice, we rewrite the symbolic variable (h, f_n) as some (fictitious) scalar variable, say δ, and symbolically execute $\lambda c : c \sqcup [\![\delta = x_n]\!]^\sharp_n (c)$ using traditional numerical analyses, e.g. interval analysis, equipped with the abstract semantics $[\![\cdot]\!]^\sharp_n$ of assignment and the abstract operator of join $\sqcup$ (specified in Sect. 2.2).

$$[\![y_p.f_n = x_n]\!]^\sharp (\bar{\mathsf{n}}, \hat{\mathsf{p}}) \triangleq \left(\left(\bigsqcup_{\hat{\mathsf{p}} \vdash y_p.f_n \Downarrow \delta} \bar{\mathsf{n}} \sqcup [\![\delta = x_n]\!]^\sharp_n (\bar{\mathsf{n}}) \right), \hat{\mathsf{p}} \right) \tag{15}$$

Note that it is not necessary to compute transfer functions for assertions involving field expressions for they are transformed beforehand by our front-end SOOT to assertions in WHILE_n or in WHILE_p. For instance, a source code `if (x.f>0) ...`, is transformed to `a = x.f; if (a>0) ...` . before our analysis.

Join and Widening The join of two facts is defined as the set of all facts that are implied independently by both. The join of $(\bar{\mathsf{n}}_1, \hat{\mathsf{p}}_1)$ and $(\bar{\mathsf{n}}_2, \hat{\mathsf{p}}_2)$ is the join of $\bar{\mathsf{n}}_1$ and $\bar{\mathsf{n}}_2$, paired with the join of $\hat{\mathsf{p}}_1$ and $\hat{\mathsf{p}}_2$.

$$(\bar{\mathsf{n}}_1, \hat{\mathsf{p}}_1) \sqcup^\sharp (\bar{\mathsf{n}}_2, \hat{\mathsf{p}}_2) = (\bar{\mathsf{n}}_1 \sqcup \bar{\mathsf{n}}_2, \hat{\mathsf{p}}_1 \cup \hat{\mathsf{p}}_2) \tag{16}$$

When computing the fixpoint, the iterates of our numerical points-to domain do not necessarily converge because of its numerical components. We perform a piecewise widening for the numerical part.

$$(\bar{\mathsf{n}}_1 \hat{\mathsf{p}}) \bigtriangledown^\sharp (\hat{\mathsf{n}}_2, \hat{\mathsf{p}}_2) = (\bar{\mathsf{n}}_1 \bigtriangledown \bar{\mathsf{n}}_2, \hat{\mathsf{p}}_1 \cup \hat{\mathsf{p}}_2) \tag{17}$$

Theorem 1 (Soundness). *The transfer functions* $[\![\cdot]\!]^\sharp : \mathsf{WHILE}_{np} \to (\mathit{NumP} \to \mathit{NumP})$*, defined in* (12), (13), (14) *and* (15)*, are sound with respect to* $[\![\cdot]\!]^\natural$*: for any statement* s *of* WHILE_{np} *and abstract state* $(\bar{\mathsf{n}}, \hat{\mathsf{p}})$ *of NumP,* $[\![s]\!]^\natural \circ \gamma''(\bar{\mathsf{n}}, \hat{\mathsf{p}}) \subseteq \gamma'' \circ [\![s]\!]^\sharp (\bar{\mathsf{n}}, \hat{\mathsf{p}})$.

Proof. We only sketch the case of $[\![x_p.f_n = y_n]\!]^\sharp$. For all $\bar{\mathsf{n}} \in \mathit{Num}^\sharp[\Delta \cup \mathit{Var}_n]$ and $\hat{\mathsf{p}} \in \mathit{Pter}^\sharp$, we prove

$$[\![x_p.f_n = y_n]\!]^\natural (\gamma''(\bar{\mathsf{n}}, \hat{\mathsf{p}})) \;\dot{\subseteq}\; \gamma''([\![x_p.f_n = y_n]\!]^\sharp (\bar{\mathsf{n}}, \hat{\mathsf{p}})) \tag{18}$$

By the Def. of $[\![x_p.f_n = y_n]\!]^\natural$ and $[\![x_p.f_n = y_n]\!]^\sharp$ and the monotony of γ_δ, it is sufficient to show for any d s.t. $\gamma_p(\hat{\mathsf{p}}) \vdash x_p.f_n \Downarrow d$,

$$[\![d = y_n]\!]^\natural_n \circ \gamma_\delta(\bar{\mathsf{n}}) \subseteq \gamma_\delta(\bar{\mathsf{n}} \sqcup [\![\delta = y_n]\!]^\sharp_n (\bar{\mathsf{n}})) \tag{19}$$

where we note $\delta = \triangleright(d)$.

By the Def. of γ_δ, it is then sufficient to prove a stronger condition:

$$\forall \sigma \in \mathtt{Ins}_\triangleright : [\![d = y_n]\!]^\natural_n \circ \gamma_n \circ [\sigma](\bar{\mathsf{n}}) \subseteq \gamma_n \circ [\sigma](\bar{\mathsf{n}}) \cup \gamma_n \circ [\sigma]([\![\delta = y_n]\!]^\sharp_n (\bar{\mathsf{n}})) \tag{20}$$

Given an instantiation σ (as defined in 4, we make two cases to conclude:

– **Case I:** σ does not map δ to d. By consequence d does not appear in $[\sigma](\bar{\mathsf{n}})$ and $[\![d = y_n]\!]_n^\sharp \circ \gamma_n \circ [\sigma](\bar{\mathsf{n}}) = \gamma_n \circ [\sigma](\bar{\mathsf{n}})$. This concludes this case.
– **Case II:** σ maps δ to d. We can then simplify the right part of (20) because $[\sigma]([\![\delta = y_n]\!]_n^\sharp(\bar{\mathsf{n}})) = ([\![d = y_n]\!]_n^\sharp \circ [\sigma](\bar{\mathsf{n}}))$. We then conclude this last case using the soundness of $[\![d = y_n]\!]_n^\sharp$.

6 Experiments

We have implemented a prototype for the abstract domain *NumP*. Below, we write NumP (in sans-serif font) for the prototype. NumP uses SOOT [33] as the front-end. It modularly combines the pointer analyses in SPARK [20], and the abstract numerical domains implemented in PPL [2].

We first implement the traditional static numerical analyzer for Java. The implementation will be denoted by Num. This is done by wrapping abstract domains in PPL. Num either skips unrecognized statements or conservatively approximates them using the "unconstraint" operators. The flow-insensitive points-to analyses are directly available in SOOT. They will be denoted by Pter subsequently.

The prototype NumP is built on Num and Pter. The input program is a set of Java classes with a class `main` indicating the entry point used for call-graph construction. NumP runs in three steps. First, it computes a global points-to graph using Pter. Then, it transforms each input class into Jimple codes using SOOT and enriches the intermediary representation with symbolic variables. At last, NumP performs a static numerical analysis of each transformed input class using an extended version of Num that takes symbolic variables into account.

In this prototype, the side effects of function calls are not considered. The output of NumP is the invariants before each Jimple statement. As experimental study, we compare NumP with its component analyses for precision and cost. Our tests were run on a 3.06 GHz Intel Core 2 Duo with 4 GB of DDR3 RAM.

6.1 `Bellman-Ford`

A case study is carried out on a small program, `Bellman-Ford`, taken from the benchmarking of `Jchord` [21][3]. Its small size (< 500 LOC in Jimple IR) allows us to run different combined analyses, including the expensive polyhedral numerical analysis and the context-sensitive points-to analysis. The objective here is to "plug in" various combinations and evaluate their precision/cost tradeoff.

It is hard to compare the precision between NumP and Num. One metric that we find reasonable is the inferred invariant number. Recall that NumP uses exactly the same transfer functions from Num for statements in WHILE_n, and is able to deal with statements that are in WHILE_{np} but are not in WHILE_n. We count the number of non-trivial invariants (an invariant is trivial if it is `true`) generated by PPL. An invariant in PPL is a conjunction of unit inequalities. In

[3] See http://code.google.com/p/jchord/source/browse/trunk/test/bench/bellman-ford/src/BellmanFord.java?r=1550.

our context, these invariants may involve symbolic variables. We count $K + 2$ times for an invariant expressed as $\{x \leq 3, y \leq 4, \delta \leq 5\}$ if δ represents K field expressions *that literally appear in the program.*

Table 2: Case study on `Bellman-Ford`

	Analysis	invariants	time (s)
Num	`intv`	984	3.54
	`poly`	1141	5.82
Pter	`spark`	0	54.45
	`geom`	0	114.24
NumP	`intv` + `spark`	1180	77.84
	`intv` + `geom`	1124	112.32
	`poly` + `spark`	1460	92.68
	`poly` + `geom`	1661	115.40

The first two columns of Tab. 2 show the instanced analyses, with `intv` denoting the interval abstract domain (`Int64_Box` from PPL), `poly` denoting the polyhedral abstract domain (`NNC_Polyhedron` from PPL), `spark` denoting the flow-insensitive, context-insensitive points-to analysis used by default in SOOT (from SPARK), and `geom` denoting the flow-insensitive, context-sensitive points-to anlaysis using geometric encoding algorithm [37] (from SPARK).

The last two columns show the number of inferred invariants and the time consumed by each analysis. The results confirm the expectation *viz.*, that the numerical analysis combined with pointer analysis infers more invariants than the numerical analysis only, and is more expensive. The best precision is obtained by polyhedral analysis combined with geometric encoding points-to analysis, which is also the most time-consuming. For this small program, the time spent by interval and polyhedral analyses is negligible compared with pointer analysis. This is because Num is intra-procedural so its complexity depends only on the length and the variable number of the program itself, whereas the points-to analysis is inter-procedural so its complexity depends on its dependent classes which are more than 10000 for this small program. Also note that the time spent by NumP is not necessarily more than the addition of its component analyses (compare `poly`, `geom` and their combined `poly` + `geom` for example). This might be due to the fact that we are using SOOT front-end which transfers programs to Jimple before each analyses.

Below, we use the invariant number and the consumed time as two performance indicators.

6.2 Dacapo

We use `Dacapo-2006-MR2` to evaluate our analysis on real-world programs. Tab. 3 gives our experimental results for the combined `intv` +`spark`. We have also tried

with polyhedral analysis in which case neither NumP nor Num is able to run over any of the benchmarks. Column 1 of the table gives the eight chosen benchmarks. In column 2, i_{Num} and i_{NumP} are the invariants numbers discovered by Num and by NumP respectively. The invariants number is the total non-trivial invariants collected from each individual method. We use

$$q_i \triangleq i_{\mathsf{NumP}}/i_{\mathsf{Num}} - 1 \tag{21}$$

as the indicator for the *precision enhancement*.

In column 3, we show the time consumed by NumP, Num and Pter. What we really care about is the time spent by NumP compared with its combined components. The *time overhead* is quantified with q_t defined as

$$q_t \triangleq t_{\mathsf{NumP}}/(t_{\mathsf{Num}} + t_{\mathsf{Pter}}) - 1 \tag{22}$$

where t_{Num}, t_{Pter} and t_{NumP} are the total time spent by the analyzers (in seconds).

Table 3: Performance test of NumP for the `Dacapo-2006-MR2` benchmark

Benchmark	Invariants Numbers			Analysis Time (s)			
	i_{Num}	i_{NumP}	q_i	t_{Num}	t_{Pter}	t_{NumP}	q_t
bloat	70238	650091	8.3	16.6	62.6	98.8	0.25
chart	76972	905011	10.8	18.5	137.6	158.1	0.01
eclipse	58377	80875	0.4	14.8	40.5	56.1	0.01
fop	69170	12354926	177.6	23.2	136.3	300.4	0.88
hsqldb	154151	3328080	20.6	30.6	277.9	345.7	0.12
jython	105775	460900	3.4	181.6	134.5	204.4	-0.35
pmd	50023	425933	7.5	15.61	120.0	140.0	0.03
xalan	109147	1050445	8.6	17.02	91.9	122.1	0.12
MEAN	86732	2407033	29.6	39.7	125.2	178.2	0.13

In Tab. 3, NumP gives an average of 29.6× precision enhancement with a time overhead of 0.13. This is meaningful: (1) Traditional Num is shown insufficient to analyze numerical properties in real-world programs because a large number of the numerical invariants involved in the program logic are not expressible by scalar variables. It is with the help of a combined pointer analysis that these alien invariants to Num may be discovered. (2) The proposed NumP has the full capability to achieve this ambition because it has little complexity overhead compared to its component analyses.

.

7 Related Work

While a large number of articles cover issues related to pointer analyses and to numerical abstractions, the program analyses where both pointers and numerics are taken into account are comparatively few.

Our work was initially inspired by Chang and Leino's congruence-closure abstract domain [6]. Their combined abstract domain extends the properties representable by a given abstract domain to schema over arbitrary terms, and not just variables. They deal with alias problem using an ad-hoc *heap succession* abstract domain while we allow to reuse off-the-shell points-to analyses.

Points-to analysis is well known, and many variants have been published (see [18] for a survey). It offers a large spectrum of tradeoffs between precision and scalability with so called *equality-based* [32], *subset-based* [1] and *flow-sensitive* [8] variations. Points-to analyses are relatively imprecise compared to more advanced shape analysis techniques, but they scale well to large programs. Most analyses that combine numerical and pointer information tend to comply with similar simple pointer analyses (TVLA shown below is clearly an exception). Logozzo's Cibai (Class Invariants By Abstract Interpretation) [22] is a modular analysis that combines a type-based pointer analysis and octagons. Sotin and Jeannet [31] extend their generic numerical analyzer Interproc to deal with programs in the presence of pointers to the stack. Miné's [24] shows the power of this simple abstraction by extending it to pointer arithmetic, union types and records of stack variables. The resulted abstraction is integrated to ASTREE [4] and is able to deal with a subset `C` program that does not have dynamic memory allocation.

The book of Simon [30] gives an extensive study of numeric analysis to avoid buffer-overflows problems in C programs. The author combines ad-hoc numerical domains and a manually refined flow-sensitive points-to analysis. The combination of Simon's work have mutual effect between the heap domain and the numeric domains. His analysis is more precise than that of Miné's and the analysis in this paper, but requires important implementation efforts compared to our modular analysis.

A more sophisticated heap abstraction is *shape analysis* [28]. The TVLA [19] framework based on shape analysis uses *canonical abstraction* to create bounded-size representations of memory states. The analyses of this family are precise and expressive. TVLA users are demanded to specify the concrete heap using first-order predicates with transitive closure, or user-defined *instrumentation predicates* like `IsNotNull`. Then TVLA automatically derives an abstract semantics based on the users' specification. The numerical abstraction of Gopan *et al.* [17] allows the integration of TVLA with existing numerical domains. The recent TVAL+ [16] uses TVLA on top of SAMPLE (Static Analysis of Multiple LanguagEs), and can be combined with any existing numerical analyses in SAMPLE. The static verifier DESKCHECK [23] combines TVLA and numerical domains. It is sufficiently precise and expressive to check quantified invariants over both heap objects and numerics. Besides the burden for users to specify the program (a problem that XISA [7] attempts to remedy), the major issue of the shape-analysis-based approaches lies in their scalability. In contrast, our experiments show our capability to run over large programs.

Pioli and Hind [26] show the mutual dependence of *conditional constant analysis* and pointer analysis. The combination is specifically designed for the condi-

tional constant analysis and is not generalized to standard numerical domains. In particular, this approach does not directly cooperate with standard numerical domains because their method relies on the particular feature of conditional constant analysis that is able to partially eliminate infeasible branches.

In a somewhat different strand of work, numerical domains have been used to enhance pointer analysis. Deutsch [14] uses a parametrised numerical domain to improve the accuracy of alias analysis in the presence of recursive pointer data structures. The key idea is to quantify the symbolic field references with integer coefficients denoting positions in data structures. This analysis is able to express properties for cyclic structures such as "for any k, the k-th element of list l of length len, is aliased to its $(k+len)$-th element". Venet [36] develops the structure called the *abstract fiber bundle* to formalize the idea of embedding an abstract numerical lattice within a symbolic structure. The structure enables the using of the large number of existing numerical abstractions to encode a broad spectrum of symbolic properties.

8 Conclusion

The primary objective of this work has been the automatic discovery of numerical invariants in Java-like programs, which are generally pointer-aware. We have proposed a methodology for combining numerical analyses and points-to analysis, developed using an approach based on concepts from abstract interpretation. In particular, we have shown how the abstract domain used in points-to analysis can be used to lift a numerical domain to encompass values stored in the heap. The new abstract domain and the accompanying transfer functions have been specified formally and their correctness proved. Moreover, the modular way in which the abstract domains are combined via some well-defined interfaces is reflected in the modular construction of a prototype implementation of the analysis framework. This modularity has enabled us to experiment with different choices for the tradeoff between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators. Concretely, the derived abstract semantics allows us to combine existing numerical domains (interval domains, polyhedra etc.) with existing points-to analyses. The modular analyzer uses PPL and SPARK and shows a clear precision enhancement with low time overhead.

Acknowledgments. The authors wish to express their gratitude to Laurent Mauborgne for his thoughtful feedback.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

2. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Technical Report 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.
3. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.
5. Peter J. Cameron. Notes on counting. Available at http://www.maths.qmw.ac.uk/pjc/notes/counting.pdf.
6. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.
7. Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *SAS*, pages 384–401, 2007.
8. Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.
9. 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, Paris, France, 1976.
10. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
11. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.
12. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
13. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.
14. A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *ICCL*, pages 2–13, 1992.
15. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.
16. Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. Tval+ : Tvla and value analyses together. In *SEFM*, pages 63–77, 2012.
17. Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.
18. Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of PASTE 2001*, pages 54–61. ACM, 2001.

19. Tal Lev-Ami and Shmuel Sagiv. Tvla: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.
20. Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.
21. Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM.
22. Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI*, pages 283–298, 2007.
23. Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, pages 71–99, 2010.
24. Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.
25. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
26. Anthony Pioli and Michael Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical report, IBM T. J. Watson Research Center, 1999.
27. Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 296–309, New York, NY, USA, 2005. ACM.
28. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.
29. Olin Shivers. Control-flow analysis of higher-order languages. Technical report, 1991.
30. A. Simon. *Value-Range Analysis of C Programs*. Springer, August 2008.
31. Pascal Sotin and Bertrand Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In *ESOP'11 : Proceedings of the 20th European Symposium on Programming*, pages 459–479, 2011.
32. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. of POPL 1996*, pages 32–41. ACM Press, 1996.
33. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.
34. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, page 13, 1999.
35. Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, July 1998.
36. Arnaud Venet. Towards the integration of symbolic and numerical static analysis. In *VSTTE*, pages 227–236, 2005.
37. Xiao Xiao and Charles Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for java. In *ISSTA*, pages 188–198, 2011.

A Proof of Soundness

A.1 Preliminaries

Notations Given $\mathsf{n} \in Num$, its *definition domain* is denoted by $dom(\mathsf{n})$. Given $\bar{\mathsf{n}} \in Num^\sharp$, its *free variable*, denoted by $FV(\bar{\mathsf{n}})$, is the union of the free variables of each formula in $\bar{\mathsf{n}}$. The space of bijective functions from A to $\widetilde{A}$ is denoted by by $A \leftrightarrow \tilde{A}$.

Definition 7 (Variable substitution). *Given $n \in \mathsf{Num}$ and a* bijective function $\sigma : dom(n) \leftrightarrow \widetilde{dom}$ *from n's definition domain to some isomorphic $\widetilde{dom}$, we define the operator of* variable substitution*, written as $[\sigma]$, to be a mapping of type $Num \to Num$ defined as*

$$[\sigma] \triangleq \lambda \mathsf{n}.(\mathsf{n} \circ \sigma^{-1}) \tag{23}$$

The definition above requires that σ be bijective, and the domain of the mapping σ be the domain of the numerical environment n. This requirement makes the operator $[\sigma]$ a bijection.

Lemma 3. *Given $\mathsf{n} \in Num$, $\sigma \in dom(n) \leftrightarrow \widetilde{dom}$, we have $[\sigma]^{-1} = [\sigma^{-1}]$*

A substitution of numerical properties is to be understood as the usual capture-avoiding substitution in lambda logic. Again, this substitution will be specified by a bijective function. Although not necessary, we require the definition domain of the specified function be exactly the same as the set of the free variables of the considered numerical property.

Definition 8 (Substitution of numerical properties). *Let $\bar{\mathsf{n}} \in Num^\sharp$ and $\sigma \in FV(\bar{\mathsf{n}}) \leftrightarrow \widetilde{FV}$ be a bijection. By abuse of language, we denote by $[\sigma]\bar{\mathsf{n}}$ the capture-avoiding substitution using σ of each of its formula.*

It can be seen that Lem. 3 holds for the overloaded $[\sigma]$ as well. For instance, let $\bar{\mathsf{n}}$ be $\{x+y<5, z<10\}$, $\sigma = \{(x,a),(y,b),(z,c)\}$, then $[\sigma]\bar{\mathsf{n}}$ is $\{a+b<5, c<10\}$. Applying $[\sigma^{-1}]$ to the latter, we immediately obtain $\bar{\mathsf{n}}$.

The following lemma states that the operation of substitution preserves the relation of valuation (defined in Sect. 2.2). For example, let $\mathsf{n} = \{(x,2),(y,3),(z,5)\}$, $\bar{\mathsf{n}} = \{x+y=z, y \le z\}$, and $\sigma = \{(x,a),(y,b),(z,c)\}$, then $\mathsf{n} \models \bar{\mathsf{n}}$ and $[\sigma]\mathsf{n} \models [\sigma]\bar{\mathsf{n}}$.

Lemma 4 (Substitution). *Given $\mathsf{n} \in Num, \bar{\mathsf{n}} \in Num^\sharp$ and a bijective σ s.t. $dom(\mathsf{n}) = dom(\sigma) = FV(\bar{\mathsf{n}})$, then $n \models \bar{\mathsf{n}} \Rightarrow [\sigma]\mathsf{n} \models [\sigma]\bar{\mathsf{n}}$.*

This lemma requires that the definition domain of n equals to the set of the free variables in $\bar{\mathsf{n}}$. To apply the lemma of substitution for the case where n has more defined variables than $\bar{\mathsf{n}}$'s free variables and $\mathsf{n} \models \bar{\mathsf{n}}$, we can restrict the definition domain of n to the free variables in $\bar{\mathsf{n}}$ so that the restricted n is a valuation of $\bar{\mathsf{n}}$. This is stated by the lemma below.

Lemma 5 (Restriction). $\mathsf{n} \models \bar{\mathsf{n}} \Rightarrow \mathsf{n}_{|FV(\bar{\mathsf{n}})} \models \bar{\mathsf{n}}$.

A.2 Proof

Proof (Proof of Thm. 1 , for the case of $x_p.f_n = y_n$). Take an arbitrary $\bar{\mathsf{n}} \in \mathit{Num}^\sharp[\Delta \cup \mathit{Var}_n]$ and an arbitrary $\hat{\mathsf{p}} \in \mathit{Pter}^\sharp$, we will prove

$$[\![x_p.f_n = y_n]\!]^\natural(\gamma''(\bar{\mathsf{n}}, \hat{\mathsf{p}})) \;\dot{\subseteq}\; \gamma''([\![x_p.f_n = y_n]\!]^\sharp(\bar{\mathsf{n}}, \hat{\mathsf{p}})) \tag{24}$$

Denote the left and the right parts of (24) `lhs` and `rhs` respectively. By the Def. of $[\![\cdot]\!]^\natural$ and $[\![\cdot]\!]^\sharp$, we have

$$\texttt{lhs} = \left(\left(\bigcup_{\gamma_p(\hat{\mathsf{p}}) \vdash x_p.f_n \Downarrow d} [\![d = y_n]\!]_n^\natural(\gamma_\delta(\bar{\mathsf{n}})) \right), \gamma_p(\hat{\mathsf{p}}) \right) \tag{25}$$

$$\texttt{rhs} = \left(\gamma_\delta \left(\bigsqcup_{\hat{\mathsf{p}} \vdash x_p.f_n \Downarrow \delta} \bar{\mathsf{n}} \sqcup [\![\delta = y_n]\!]_n^\sharp(\bar{\mathsf{n}}) \right), \gamma_p(\hat{\mathsf{p}}) \right) \tag{26}$$

Take an arbitrary d s.t. $\gamma_p(\hat{\mathsf{p}}) \vdash x_p.f_n \Downarrow d$ and let $\delta = \triangleright(d)$. We will prove

$$[\![d = y_n]\!]_n^\natural \circ \gamma_\delta(\bar{\mathsf{n}}) \subseteq \gamma_\delta(\bar{\mathsf{n}} \sqcup [\![\delta = y_n]\!]_n^\sharp(\bar{\mathsf{n}})) \tag{27}$$

By the Def. of γ_δ, it suffices to prove a stronger condition:

$$\forall \sigma \in \texttt{Ins}_\triangleright : [\![d = y_n]\!]_n^\natural \circ \gamma_n \circ [\sigma](\bar{\mathsf{n}}) \subseteq \gamma_n \circ [\sigma](\bar{\mathsf{n}} \sqcup [\![\delta = y_n]\!]_n^\sharp(\bar{\mathsf{n}})) \tag{28}$$

Let the left and the right parts of (28) denoted by $\texttt{lhs}'$ and $\texttt{rhs}'$ respectively. Take an arbitrary $\sigma \in \texttt{Ins}_\triangleright$, let $\sigma \triangleq \{(\delta_i, d_i)\}_{1 \le i \le |\Delta|}$. Take an arbitrary $\mathsf{n} \in \mathit{Num}[D \cup \mathrm{Var}_n]$ satisfying

$$\mathsf{n} \in \texttt{lhs}' \tag{29}$$

we want to show

$$\mathsf{n} \in \texttt{rhs}' \tag{30}$$

By Eq. (29) and the correctness of $[\![\cdot]\!]_n^\sharp$, we have $\xrightarrow{\mathit{Num}} (d = y_n)(\mathsf{n}) \models [\sigma]\bar{\mathsf{n}}$. Below we will write

$$\mathsf{n}[d \mapsto y_n]$$

namely n updated with $d = y_n$, for $\xrightarrow{\mathit{Num}} (d = y_n)(\mathsf{n})$. We have

$$\mathsf{n}[d \mapsto y_n] \models [\sigma]\bar{\mathsf{n}} \tag{31}$$

Thus, by Lem. 4, we have[4]

$$[\sigma^{-1}](\mathsf{n}[d \mapsto y_n]) \models \bar{\mathsf{n}} \tag{32}$$

We continue the proof following whether σ maps δ to d.

[4] Although σ is not a bijection, it is guaranteed to be injective. Thus σ^{-1} is still well-defined.

– **Case I :** $(\delta, d) \in \sigma$. Since $d \in dom(\sigma^{-1})$, (32) implies

$$([\sigma^{-1}]\mathsf{n})[\delta \mapsto y_n] \models \bar{\mathsf{n}} \tag{33}$$

By the soundness of $[\![\cdot]\!]^\sharp_n$, we have

$$[\sigma^{-1}]\mathsf{n} \models [\![\delta = y_n]\!]^\sharp_n \, \bar{\mathsf{n}} \tag{34}$$

– **Case II** $(\delta, d) \notin \sigma$. Since $d \notin dom(\sigma^{-1})$, (32) implies

$$[\sigma^{-1}]\mathsf{n} \models \bar{\mathsf{n}} \tag{35}$$

due to Lem. 5.

Combining the two cases, we obtain (30) following Lem. 4.