Modularly Combining Numeric Abstract Domains with Points-to Analysis, and a Scalable Static Numeric Analyzer for Java

Zhoulai Fu

Modularly Combining Numeric Abstract Domains with Points-to Analysis, and a Scalable Static Numeric Analyzer for Java

Zhoulai Fu *

Université de Rennes 1 – INRIA, France

Abstract. This paper contributes to a new abstract domain that combines static numeric analysis and points-to analysis. One particularity of this abstract domain lies in its high degree of modularity, in the sense that the domain is constructed by reusing its combined components as black-boxes. This modularity dramatically eases the proof of its soundness and renders its algorithm intuitive. We have prototyped the abstract domain for analyzing real-world Java programs. Our experimental results show a tangible precision enhancement compared to what is possible by traditional static numeric analysis, and this at a cost that is comparable to the cost of running the numeric and pointer analyses separately.

1 Introduction

Static numeric analysis – that approximates values of scalar variables and their relationship – has drawn on a rich body of techniques including abstract domains of intervals [9], polyhedron [13] and octagons [24] etc. 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, when extending numeric analyses to heap-manipulating programs we are immediately faced with the issues that pointers introduce *aliases* which make program reasoning difficult because understanding the communication between numeric properties and dynamic data structures is needed. This gives rise to *the problem of combining static numeric analysis and heap analysis*.

The combination of the two analyses has been studied, but the solutions proposed so far tend to be complex to implement or impractical to analyze large programs. For example, Simon [27] shows how to combine *ad hoc* numeric abstract domains with manually refined flow-sensitive points-to analyses. His combination approach requires extensive experiences and intimate familiarity with the abstract domains themselves, thereby hard to implement. Miné's abstraction [23], by contrast, is designed to be modular. The purpose was to lift

* The research leading to these results has received financial help from AX – L'Association des Anciens Élèves et Diplômés de l'École polytechnique. 5, rue Descartes 75005 PARIS.

existing abstract domains in ASTREE [3] developed with several man-years to cope with pointer-aware programs. Reusing existing components as modules is particularly important in that context. However, Miné's framework is based on type-based pointer analysis, which is cheap but too coarse by its nature. This prohibits the general practicability of the Miné's analysis. At the other extreme, shape-analysis [26] based approaches come with sophisticated pointer analyses and can indeed infer non-trivial properties. However, analyses that are based on shape abstraction can hardly (see [5, 32] for exceptions) run on large programs.

Different from the work mentioned above, our objective is to develop a combined analysis satisfying the following requirements:

- **Modular design:** The combined analysis should enable the reusing of existing analyses that have been developed since decades. The construction of the combined analysis should only depend on the interfaces, not the specific implementations, of its components.
- **Scalability:** We are seeking a tool that runs on codes of hundreds of thousands of lines. We examine the feasibility of our analysis over moderate and large sized benchmarks, and ensure that the combined analysis only presents small complexity overhead compared with its component analyses.
- **Precision:** Although the query of scalability inevitably demands a sacrifice on precision, we inspect that the combined analysis has to be, at least, as precise as its components.

The core contribution of this work is a theoretical foundation that combines in a generic manner

- an abstract domain dedicated to static numeric analysis of programs without allocations, and
- an abstract domain for points-to static analysis.

On the practical side, we have implemented the abstract domain, using the Java Optimization Framework SOOT [29] as the front-end, and relying on the abstract domains from existing static analysis libraries such as the Parma Polyhedra Library PPL [1] and the SOOT Pointer Analysis Research Kit SPARK [20]. This prototype analyzer, called NumP, has been run on *all* 11 programs in the Dacapo-2006-MR2 [4] benchmark suite. The suite is composed of moderate and large sized program with rich object behaviors and demanding memory system requirements. Our experiments confirm that the combined analysis is feasible even for large-sized programs and that it discovers significantly more program properties than what is possible by pure numeric analysis, and this at a cost that is comparable to the cost of running the numeric and pointer analysis separately.

1.1 Organization of the paper

The interfaces of traditional numeric and pointer analyses 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 modeled language and its concrete semantics. The abstract

domain and its operators are presented in Sect. 5. Experimental results are shown in Sect. 6. Finally, we compare our analysis with related work and conclude in Sect. 7 and 8.

The formal underpinning and semantic correctness of the combination technique are presented in the author's Ph.D. thesis [17].

2 Analysis Interfaces

This section is define the interfaces of two existing analysis, static numeric analysis and points-to analysis.

General notation. 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 $m(x) = \perp$.

Syntactical notations. Primary data types include: scalar numbers in $\mathbb{I}$, where $\mathbb{I}$ can be integers, rationales or reals; and references (or pointers) in *Ref*. Primary syntactical entities include the universe of *local variables* and *fields*. They are denoted by *Var* and *Fld* respectively. An *access path* is either a variable or a variable followed by a sequence of fields. The universe of access paths is denoted by *Path*. We subscript Var_τ, Field_τ, or $Path_\tau$ with $\tau \in \{n, p\}$ to indicate their types as a scalar number or a reference, respectively. The elements in these sets can also be sub-scripted with types. The types will be omitted if they are clear from context.

We use Imp_n to refer to the basic statements only involving numeric variables and use the meta-variables s_n to range over these statements. Similarly, we let Imp_p be the statements that only use pointer variables and let s_p range over these statements. Below we list the syntactical entities and meta-variables used to range over them.

$k \in \mathbb{I}$	scalar numbers
$r \in Ref$	concrete references
$x_\tau, y_\tau \in \text{Var}_\tau$	numeric/pointer variables
$f_\tau, g_\tau \in \text{Field}_\tau$	numeric/pointer fields
$\boldsymbol{u}_\tau, \boldsymbol{v}_\tau \in Path_\tau$	numeric/pointer access paths
$s_n \in \mathsf{Imp}_n$	$x_n = k \mid x_n = y_n \mid x_n = y_n \diamond z_n \mid x_n \bowtie y_n$
$s_p \in \mathsf{Imp}_p$	$x_p = \texttt{new} \mid x_p = y_p.f_p \mid x_p = y_p \mid x_p.f_p = y_p$

where $\diamond \in \{+, -, *, /\}$, and $\bowtie$ is an arithmetic comparison operator.

2.1 Static numeric analysis

Static numeric analysis can be modeled as an abstract interpretation of Imp_n.

We use the term *numeric property* [22] for any conjunction of formula in a certain theory of arithmetic. For example, the numeric property $\{x^2 + y^2 \leq$

$1, \quad x \leq 0, \quad y \leq 0\}$ is composed of the conjunction of three arithmetic formulas As usual, an *environment* maps variables to their values. We consider *numeric environments*:

$$Num \triangleq Var_n \to \mathbb{I}_\perp \tag{1}$$

The relationship between an environment and a property can be formalized by the concept of *valuation*. We say that n is a valuation of $\mathsf{n}^\sharp$, denoted by

$$\mathsf{n} \models \mathsf{n}^\sharp \tag{2}$$

if $\mathsf{n}^\sharp$ becomes a tautology after each of its free variables, if any, has been replaced by its corresponded value in n.

Definition 1 (Interface of the traditional numeric analyzer).

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

The concrete numeric domain *and the* abstract numeric domain *for the language* Imp_n *are* $\wp(Num)$ *and* $Num^\sharp$ *respectively. They are related by the* concretization function $\gamma_n : Num^\sharp \to \wp(Num)$ *defined by* $\gamma_n(\mathsf{n}^\sharp) = \{n \in Num \mid n \models \mathsf{n}^\sharp\}$.

The partial order $\sqsubseteq$ *is consistent with the monotonicity of* γ_n, i.e., $\mathsf{n}_1^\sharp \sqsubseteq \mathsf{n}_2^\sharp$ *implies* $\gamma_n(\mathsf{n}_1^\sharp) \subseteq \gamma_n(\mathsf{n}_2^\sharp)$. *For each statement* s_n *of* Imp_n, *the concrete semantics is given by a standard transfer function* $[\![s_n]\!]_n^\natural \in \wp(Num) \to \wp(Num)$. *The abstract semantics* $[\![\cdot]\!]_n^\sharp$ *satisfies the soundness condition:*

$$[\![\cdot]\!]_n^\natural \circ \gamma_n \subseteq \gamma_n \circ [\![\cdot]\!]_n^\sharp \tag{3}$$

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 the soundness of* $\bigtriangledown$ *is specified in [10].*

2.2 Pointer analysis

Pointer analysis can be modeled as an abstract interpretation of Imp_p.

Let *Pter* be the set of concrete states in Imp_p. Traditionally, a state $\mathsf{p} \in Pter$ is a pair of environment and heap. We write p to range over them.

$$\mathsf{p} \in Pter \triangleq (Var_p \to Ref_\perp) \times ((Ref \times Fld_p) \to Ref_\perp) \tag{4}$$

The essence of *pointer analysis* is the process of heap disambiguation, *i.e.*, the analysis partitions *Ref* into a finite set H and then summarizes the run-time pointer relations via elements h in H. The process is based on the *naming scheme*.

Definition 2. *The* naming scheme *is a mapping from concrete references to their names in* H. *The names used by the naming scheme of a pointer analysis are called* abstract references *or* abstract locations.

$$\rhd \in Ref \to H \tag{5}$$

We say $r \in Ref$ is abstracted by $h \in H$ if $r \rhd h$. It is required that the memory regions abstracted by different abstract references have no common concrete reference. $\forall h_1, h_2 \in H, h_1 \neq h_2 \Rightarrow \rhd^{-1}(h_1) \cap \rhd^{-1}(h_2) = \emptyset$.

This paper considers points-to analysis [15] that is widely used in heap analysis. The lattice used in the points-to abstract domain is commonly called *points-to graph*. This graph has two kinds of arcs, the unlabeled arcs from a variable to an abstract reference and the labeled arcs between abstract references that are labeled by a field. The abstract domain used in points-to analysis is a set of points-to graphs, denoted by $Pter^\sharp$.

$$Pter^\sharp \triangleq (Var_p \to \wp(H)) \times ((H \times Fld_p) \to \wp(H)) \tag{6}$$

Remark 1. Points-to analysis is based on a naming scheme that is flow independent. In other words, a given analysis pass of points-to analysis allows for a unique naming scheme, whatever the abstractions of the heap. It is worth noting that this property on the naming scheme is respected by all variants of points-to analysis (including flow-sensitive points-to analysis). In this presentation, we use a typical naming scheme to name heap elements after the program point of the statement that allocates them.

Definition 3 (Interface of traditional points-to analyzer).

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

The concrete domain and the abstract domain of points-to analysis are denoted by $\wp(Pter)$ and $Pter^\sharp$ respectively. They are related by a monotone concretization function *$\gamma_p : Pter^\sharp \to \wp(Pter)$. The concrete semantics is interfaced by a standard transfer function $[\![\cdot]\!]_p^\natural \in \wp(Pter) \to \wp(Pter^\sharp)$. The abstract semantics $[\![\cdot]\!]_p^\sharp \in Pter^\sharp \to Pter^\sharp$ is provided by a static numeric analyzer. This analyzer is assumed sound:*

$$[\![\cdot]\!]_p^\natural \circ \gamma_p \subseteq \gamma_p \circ [\![\cdot]\!]_p^\sharp \tag{7}$$

3 Combining Points-to and Numeric Analysis: Intuition

This section presents the intuition behind the technique of combining points-to analyses and numeric analyses. The idea is to use the names computed by the points-to analysis to create *summarized variables* that represent the numeric values stored at particular heap locations.

Example 1. Consider the Java snippet in Listing 1.1. 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.*, the 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 in Listing 1.1) creates an object of class `Pos` or `Neg` according to whether n is positive or not. In both cases, *data.val* is assigned to the absolute value 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 links it to the list created by the precedent iteration.

Below we show how we infer the following properties at the end of the program (l. 21).

- **Prop1** Each list element of 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: $buf[*] \in [-9, 7]$.

```
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  δ_{h1,[*]} ≐ -9;
2  δ_{h1,[*]} ≐ 7;
3  δ_{h1,[*]} ≐ 3;
4  δ_{h1,[*]} ≐ -5;
5  idx = 0;
6  while (?){
7    n ≐' δ_{h1,[*]};
8    if (n > 0)
9      δ_{h2,val} ≐ n;
10     δ_{h3,val} ≐ n;
11   else
12     δ_{h2,val} ≐ -n;
13     δ_{h3,val} ≐ -n;
14   idx = idx + 1;
15 }
```

Listing 1.2: Semantics actions

Fig. 1: 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`.

We start with a *flow-insensitive points-to analysis*. A single points-to graph for the whole program can be obtained (Fig. 2). Semantically, the points-to graph

disambiguates the heap by telling what must not alias. We derive a summarized variable $\delta_{h,val}$ for each pair of heap location h and field val. The key point is, numeric values bound to syntactically distinct summarized variables are guaranteed to be stored at different concrete heap locations. In line with the semantics of points-to graph, the analysis of the program in Listing 1.1 can be treated as an *extended* numeric analysis. This analysis is called "extended" because it not only deals with scalar variables, but also deals with summarized variables.

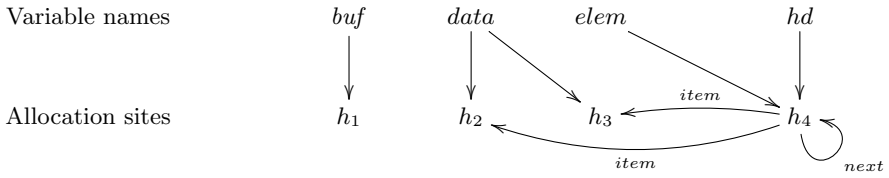

Fig. 2: A flow-insensitive points-to graph for the program in Listing 1.1.

Listing 1.2 illustrates the semantics actions taken by our analysis. From l. 1 to l. 4, the summarized variable $\delta_{h_1,[*]}$ is updated with -9, 7, 3 and -5 successively. Since more than one run-time heap locations of the array *buf* can be associated with $\delta_{h_1,[*]}$, the semantics action is a *weak update* (denoted by $\doteq$), *i.e.*, *accumulating* values rather than *overwriting* them. The semantics action at l. 7 assigns the summarized variable $\delta_{h_1,[*]}$ to the scalar variable n. Note again that this abstract semantics should be distinguished from the abstract semantics of assignment in traditional numeric domain. This is because we should not establish a numeric relation between $\delta_{h_1,[*]}$ and n as in traditional static numeric analysis. Here we use $\doteq'$ to make a distinction. Intuitively, the assignment of $\delta_{h_1,[*]}$ to n should be abstracted as assigning the possible values of $\delta_{h_1,[*]}$ to n without coupling $\delta_{h_1,[*]}$ and n. The rest of the semantics actions in the listing should be clear now. The assignments to scalar variables at l. 5 and l. 14 are the same as in traditional numeric domains. The assignments at l. 9, 10, 12, 13 are weak update to $\delta_{h_2,val}$ and $\delta_{h_3,val}$ since both h_2 and h_3 are pointed to by the variable *data* following the points-to graph.

By performing the extended interval analysis, we are able to infer these invariants at the end of the program: $\delta_{h_2,val} \in [0,9] \wedge \delta_{h_3,val} \in [0,9]$ and $\delta_{h_1,[*]} \in [-9,7]$, which imply **Prop1** and **Prop2** respectively.

Remark 2. The compelling part of this approach should not be the semantics actions presented so far, but the way that they can be constructed by an interplay between traditional numeric domains and points-to analysis. The advantage of this approach is that this interplay does not requires knowledge beyond the interfaces of the components in question. As demonstrated by our implementation of the analysis and its experimental results, this approach allows for direct access to many existing abstract domains including their join, widening and narrowing operators which are known difficult to implement.

4 The Language and its Concrete Semantics

This paper focuses on how to deal with language Imp_{np}. The statements in Imp_{np} include those in Imp_n and Imp_p, and two more statements in the forms of $y_p.f_n = x_n$ and $x_n = y_p.f_n$. We write s_{np} to range over Imp_{np}.

$$s_{np} ::= s_n \mid s_p \mid y_p.f_n = x_n \mid x_n = y_p.f_n \tag{8}$$

A concrete state in Imp_{np} can be regarded as a pair of an environment and a heap

$$\begin{aligned} State = & \overbrace{(Var_n \to \mathbb{I}_\perp) \times (Var_p \to Ref_\perp)}^{Env} \\ & \times \underbrace{((Ref \times Fld_n) \to \mathbb{I}_\perp) \times ((Ref \times Fld_p) \to Ref_\perp)}_{Heap} \end{aligned} \tag{9}$$

We can turn this domain into an isomorphic shape

$$State \triangleq Num[(Ref \times Fld_n) \cup Var_n] \times Pter \tag{10}$$

where $Num[(Ref \times Fld_n) \cup Var_n]$ extends Num to $(Ref \times Fld_n) \cup Var_n) \to \mathbb{I}_\perp$.

Remark 3. The isomorphism consists of a crucial step. It prepares the re-use of the abstract pointer values when extending the numeric domains to cover properties about heap values.

Regarding states as (10) allows us to express the concrete semantics of Imp_{np} via those of Imp_n and Imp_p. As a shortcut, we set

$$D = Ref \times Fld_n \tag{11}$$

and use meta variable d to range over the pairs in D. In Fig. 3, we show the structural operational semantics (SOS) of Imp_{np}, denoted by $\longrightarrow^\natural$. It is expressed by $\xrightarrow{Pter}$ and $\xrightarrow{Num}$ (with $\xrightarrow{Num}$ in the figure extended over $D \cup Var_n$).

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

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

Fig. 3: Structural Operational semantics $\longrightarrow^\natural : \mathsf{Imp}_{np} \to \wp(State \times State)$

We use the lifting of $\longrightarrow^\natural$ to the powerset $\wp(State)$. as the collecting semantics of Imp_{np}, denoted as

$$[\![\cdot]\!]^\natural \triangleq \lambda s : \mathsf{Imp}_{np}.\mathsf{post}[\longrightarrow^\natural(s)] \tag{12}$$

5 The abstract domain

A state in our proposed abstract domain is a pair $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$, where $\mathsf{n}^\sharp$ is a numeric property expressed via scalar variables of Var_n and summarized variables (see below) of the set $H \times Fld_n$; the element $\mathsf{p}^\sharp$ is a lattice of $Pter^\sharp$, namely, a points-to graph in our context.

Definition 4 (Summarized variable). *A summarized variable is a pair of an abstract reference $h \in H$ and a numeric field $f_n \in Fld_n$. The set of summarized variables is denoted by Δ.*

$$\Delta \triangleq H \times Fld_n \tag{13}$$

We will use the meta-variable δ to range over the pairs in Δ, or we write δ_{h,f_n} to indicate the summarized variable corresponding to (h, f_n).

Definition 5 (The abstract domain $NumP^\sharp$). *The abstract domain $NumP^\sharp$ is defined to be*

$$NumP^\sharp \triangleq Num^\sharp[\Delta \cup Var_n] \times Pter^\sharp \tag{14}$$

Below, we specify the concretization function. It consists of an essential step before defining and proving the correctness of the abstract operators on $NumP^\sharp$.

Revisit the example in Sect. 3. We have obtained the state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ at the end of the program, with

$$\mathsf{n}^\sharp = \{\delta_{h_2,val} \in [0,9], \delta_{h_3,val} \in [0,9], \delta_{h_1,[*]} \in [-9,7]\} \tag{15}$$

and $\mathsf{p}^\sharp$ is the points-to graph specified in Fig. 2. A concrete state $(\mathsf{n}, \mathsf{p}) \in State$ is in the concretization of $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ if for any reference r,

- we have $\mathsf{n}(r, val) \in [0,9]$ as long as r is abstracted by h_2, *i.e.*, $r \rhd h_2$, and
- we have $\mathsf{n}(r, val) \in [0,9]$ as long as r is abstracted by h_3, *i.e.*, $r \rhd h_3$, and
- we have $\mathsf{n}(r, [*]) \in [-9,7]$ as long as r is abstracted by h_1, *i.e.*, $r \rhd h_1$

and p has to be a concrete state abstracted by $\mathsf{p}^\sharp$, *i.e.*, $\mathsf{p} \in \gamma_p(\mathsf{p}^\sharp)$. By abuse of language, we have treated the array index $[*]$ above as an aggregate numeric field. In other words, we say (n, p) is in the concretization of $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ if n is in the concretization of all $\mathsf{n}^{\sharp\prime}$ that is the numeric property $\mathsf{n}^\sharp$ with each of its summarized variables δ substituted by some d of $Ref \times Fld_n$ (namely D) that satisfies $\rhd(d) = \delta$ (with $\rhd$ extended by taking care of numeric fields).

Definition 6 (Instantiation). *Let $\rhd$ be naming scheme that is extended from $Ref \to H$ to $Ref \times Fld_n \to H \times Fld_n$. We define the space of instantiation as a set of mappings from Δ to D.*

$$\mathtt{Ins}_\rhd \triangleq \{\sigma : \Delta \to D \mid \sigma(h, f_n) = (r, g_n) \Rightarrow h = \rhd(r) \wedge f_n = g_n\} \tag{16}$$

Definition 7. *The concretization function* γ_{np} *of* $NumP^\sharp \to \wp(State)$ *is defined as*

$$\gamma_{np}(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \{(\mathsf{n}, \mathsf{p}) \mid \mathsf{p} \in \gamma_p(\mathsf{p}^\sharp) \wedge \forall \sigma \in \mathtt{Ins}_\triangleright : \mathsf{n} \in \gamma_n \circ [\sigma](\mathsf{n}^\sharp)\} \tag{17}$$

where we denote by $[\sigma]$ *the capture-avoiding substitution operator that replaces all the free occurrences of* δ *in* $\mathsf{n}^\sharp \in Num^\sharp[\Delta \cup Var_n]$ *with* $\sigma(\delta)$.

Example 2. Consider the following program:

```
1  List hd = null, tmp;
2  int i;
3  for (i = -17; i < 42; i++){
4    List tmp = new List();   // allocation site h
5    tmp.val = i;
6    tmp.next = hd;
7    hd = tmp;
8  }
```

A list of integers ranging from -17 to 41 is stored iteratively on the heap. At each iteration, a memory cell bound to variable *tmp* is allocated. The cell consists of a numeric field *val* and a reference field *next*. The head of the list is always pointed to by the variable *hd*.

The abstract memory state computed at the end of program is given by

$$(\mathsf{n}^\sharp, \mathsf{p}^\sharp) = \left(\{\delta_{h,val} \in [-17, 41], i = 42\} \qquad \begin{matrix} \text{tmp} \\ \text{hd} \end{matrix} \longrightarrow \text{h} \circlearrowright_{next} \right) \tag{18}$$

5.1 Transfer functions

Let $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ be a state of $NumP^\sharp$. We are concerned with how it should be updated by statements of Imp_{np}.

Transfer function for s_n It is sound to assume that assignments or assertions of numeric variables have no effect on the heap. If s_n is an assignment in Imp_n, it can be treated in the same way as in traditional numeric analysis using its abstract transfer function $[\![\cdot]\!]_n^\sharp$ (as specified in Sect. 2.1).

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

$$[\![s_n]\!]^\sharp (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq [\![s_n]\!]_n^\sharp \, \mathsf{n}^\sharp, \mathsf{p}^\sharp \tag{19}$$

If s_n is an assertion in Imp_n, $\mathsf{p}^\sharp$ may be refined. For example, consider the *compound statement*[1] `if (a > 0) p = q` where p and q are reference variables and a is a numeric variable. Although it should be possible to perform a dead-code elimination using inferred numeric relations, similar to Pioli's conditional constant propagation [25], we still use the Eq. (19) for the ease of implementation.

[1] 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.

Transfer function for s_p It is also sound to assume that s_p has no effect upon $\mathsf{n}^\sharp$. Yet the reasoning is different from the above case. For example, if $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ is the state shown on Eq. (18), how can we tell whether an assignment of pointers operation modifies $\mathsf{n}^\sharp$ or not? Recall that the intended semantics of $\delta_{h,val} \to [17, 41]$ is that every value stored in each (r, val) satisfying $\rhd(r) = h$ must be in the range of $[-17, 41]$. That is to say, $\mathsf{n}^\sharp$ represents a fact about the *numeric content* stored in the corresponding concrete references. Since a pointer assignment can by no means modify any numeric values stored in the heap, the algorithm to update $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ with s_p can be written as:

$$[\![s_p]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \mathsf{n}^\sharp, [\![s_p]\!]^\sharp_p\,\mathsf{p}^\sharp \tag{20}$$

Transfer function of $y_p.f_n = x_n$ Consider an assignment $y_p.f_n = x_n$ with y_p pointing to $h \in H$. We regard $y_p.f_n = x_n$ as an weak update to summarized variable δ_{h,f_n}, That is, the field f_n of *one of* the concrete objects represented by h is to be updated with the value of x_n, while the other concrete objects represented by h remain unchanged. This effect can be approximated by $\lambda\mathsf{n}^\sharp.\mathsf{n}^\sharp \sqcup [\![\delta_{h,f_n} = x_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)$. Below, we write

$$\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta \tag{21}$$

if δ is associated with (h, val) and y_p points to h. The transfer function of $y_p.f_n = x_n$ can be modeled by joining the effects of weak update of all δ by x_n such that $\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta$.

$$[\![y_p.f_n = x_n]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \left(\left(\bigsqcup_{\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta} \mathsf{n}^\sharp \sqcup [\![\delta = x_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)\right), \mathsf{p}^\sharp\right) \tag{22}$$

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 Imp_n or in Imp_p. For instance, a source code `if (x.f > 0) ...`, is transformed to `a = x.f; if (a > 0) ...` before our analysis.

Transfer function of $x_n = y_p.f_n$ Consider the snippet

```
a = x.f; b = y.f; if (a < b) {...}
```

Assume that $\mathsf{p}^\sharp \vdash x.f \Downarrow \delta$ and $\mathsf{p}^\sharp \vdash y.f \Downarrow \delta$. It is tempting, but wrong, to abstract the semantics of $a = x.f$ (resp. $b = y.f$) as $[\![a = \delta]\!]^\sharp_n$ (resp. $[\![b = \delta]\!]^\sharp_n$) following which the analysis would incorrectly argue that the `if` branch can never be reached.

This issue was carefully studied and solved by Gopan *et al.* [18]. The authors showed that it would be wrong to correlate a summarized dimension δ to a non-summarized dimension x_n even if the former is assigned to the later; they argued that the correct way to assign a summarized dimension δ to a non-summarized

dimension x_n takes three steps: first, copy the summarized dimension δ to a fresh δ', and then relate x_n with δ' using traditional abstract semantics for assignment. Finally, the newly introduced dimension δ' has to be removed. Intuitively, the resulting abstract value keeps the possible (abstract) values of δ without being correlated with it. Gopan *et al.* have introduced four non-standard operators, in particular, "drop" that removes dimensions, and "expand" that copies dimensions. We use

$$[\![x_n = y_p.f_n]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \bigsqcup_{\mathsf{p}^\sharp \vdash y_p.f_n \Downarrow \delta} G(x_n, \delta)\ \mathsf{n}^\sharp, \quad \mathsf{p}^\sharp \tag{23}$$

where Gopan's operator $G(x_n, \delta)$ is the composition of the three steps described above:

$$G(x_n, \delta) \triangleq \lambda \mathsf{n}^\sharp.\, \mathrm{drop}^\sharp_{\delta'} \circ [\![x_n = \delta']\!]^\sharp_n \circ \mathrm{expand}^\sharp_{\delta,\delta'}\ \mathsf{n}^\sharp \tag{24}$$

Above, we assume dimension δ' does not belong to the dimensions of $\mathsf{n}^\sharp$ in question.

Example 3. Let $\mathsf{n}^\sharp = \delta \to [0,1]$. Even if we use a relational domain like polyhedral analysis, only $G(x,\delta)\mathsf{n}^\sharp = x \to [0,1], \delta \to [0,1]$ can be obtained, while traditional numeric domains would establish a relationship between x and δ.

Theorem 1 (Soundness). *The transfer functions* $[\![\cdot]\!]^\sharp : \mathsf{Imp}_{np} \to (\mathit{NumP}^\sharp \to \mathit{NumP}^\sharp)$*, defined in* (19), (20), (22) *and* (23)*, are sound with respect to* $[\![\cdot]\!]^\natural$*: for any statement* s *of* Imp_{np} *and abstract state* $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ *of* $\mathit{NumP}^\sharp$*,* $[\![s]\!]^\natural \circ \gamma_{np}(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \subseteq \gamma_{np} \circ [\![s]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$.

We give a proof sketch for the case of $[\![x_p.f_n = y_n]\!]^\sharp$. It is important to note that the soundness of the theorem is based on the soundness hypotheses of $[\![\cdot]\!]^\sharp_n$ and $[\![\cdot]\!]^\sharp_p$. The combined analysis is sound as long as its component analyses are.

Proof. For all $\mathsf{n}^\sharp \in \mathit{Num}^\sharp[\Delta \cup \mathit{Var}_n]$ and $\mathsf{p}^\sharp \in \mathit{Pter}^\sharp$, we prove

$$[\![x_p.f_n = y_n]\!]^\natural\,(\gamma_{np}(\mathsf{n}^\sharp, \mathsf{p}^\sharp)) \ \dot{\subseteq}\ \gamma_{np}([\![x_p.f_n = y_n]\!]^\sharp\,(\mathsf{n}^\sharp, \mathsf{p}^\sharp)) \tag{25}$$

By the definitions 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 such that $\gamma_p(\mathsf{p}^\sharp) \vdash x_p.f_n \Downarrow d$, we have

$$[\![d = y_n]\!]^\natural_n \circ \gamma_\delta(\mathsf{n}^\sharp) \subseteq \gamma_\delta(\mathsf{n}^\sharp \sqcup [\![\delta = y_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)) \tag{26}$$

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

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

$$\forall \sigma \in \mathtt{Ins}_\triangleright : [\![d = y_n]\!]^\natural_n \circ \gamma_n \circ [\sigma](\mathsf{n}^\sharp) \subseteq \gamma_n \circ [\sigma](\mathsf{n}^\sharp) \cup \gamma_n \circ [\sigma]([\![\delta = y_n]\!]^\sharp_n\,(\mathsf{n}^\sharp)) \tag{27}$$

Given an instantiation σ (as defined in Eq. (6)), we make two cases to conclude:

– **Case I:** σ does not map δ to d. By consequence d does not appear in $[\sigma](\mathsf{n}^\sharp)$ and $[\![d = y_n]\!]^\natural_n \circ \gamma_n \circ [\sigma](\mathsf{n}^\sharp) = \gamma_n \circ [\sigma](\mathsf{n}^\sharp)$. This concludes this case.
– **Case II:** σ maps δ to d. We can then simplify the right part of (27) because $[\sigma]([\![\delta = y_n]\!]^\sharp_n(\mathsf{n}^\sharp)) = ([\![d = y_n]\!]^\sharp_n \circ [\sigma](\mathsf{n}^\sharp))$. We then conclude this last case using the soundness of $[\![d = y_n]\!]^\sharp_n$.

5.2 Join and widening

The join of two facts is defined as the set of all facts that are implied independently by both. Thanks to our hypothesis of flow independent naming scheme (in Sect. 2.2), the join and widening of $NumP^\sharp$ are easy to define: we just have to compute the join (or widening) component wise. Then, if a concrete state (n, p) is in $\gamma_{np}(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp_1)$ or $\gamma_{np}(\mathsf{n}^\sharp_2, \mathsf{p}^\sharp_2)$, it is also in the concretization of $(\mathsf{n}^\sharp_1 \sqcup \mathsf{n}^\sharp_2, \mathsf{p}^\sharp_1 \cup \mathsf{p}^\sharp_2)$. Thus the join of $(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp_1)$ and $(\mathsf{n}^\sharp_2, \mathsf{p}^\sharp_2)$ is the join of $\mathsf{n}^\sharp_1$ and $\mathsf{n}^\sharp_2$, paired with the join of $\mathsf{p}^\sharp_1$ and $\mathsf{p}^\sharp_2$ (Sect. 2). The case for widening is similar.

$$(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp_1) \sqcup^\sharp (\mathsf{n}^\sharp_2, \mathsf{p}^\sharp_2) = (\mathsf{n}^\sharp_1 \sqcup \mathsf{n}^\sharp_2, \mathsf{p}^\sharp_1 \cup \mathsf{p}^\sharp_2) \tag{28}$$

$$(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp_1) \bigtriangledown^\sharp (\mathsf{n}^\sharp_2, \mathsf{p}^\sharp_2) = (\mathsf{n}^\sharp_1 \bigtriangledown \mathsf{n}^\sharp_2, \mathsf{p}^\sharp_1 \cup \mathsf{p}^\sharp_2) \tag{29}$$

5.3 Constraint system with a flow-insensitive points-to analysis

In our implementation, we use a flow-insensitive points-to analysis as a pre-analysis step. It is worth nothing that using flow-insensitive variant does not cause any soundness issue. This is because the soundness of our analysis is based on the soundness of its component numeric domains and pointer analysis; taking the flow-insensitive points-to graph during all propagation can be modeled as an analysis that is initialized with a set that is larger than the least fix point of a flow-sensitive analysis, and propagates in the style of `skip`, which satisfies the soundness requirement for the pointer analysis component.

Let $F^\sharp(s) \triangleq \lambda \mathsf{n}^\sharp.\mathsf{fst} \circ [\![s]\!]^\sharp(\mathsf{n}^\sharp, \mathsf{p}^\sharp_{fi})$, where $\mathsf{p}^\sharp_{fi}$ is the flow-insensitive points-to graph, and fst is the operator that extracts the first element from a pair of components. We use the following constraint system that operates on numeric lattice $\mathsf{n}^\sharp$ only (rather than on $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ pair):

$$\overline{\mathsf{n}^\sharp}[l] \sqsupseteq F^\sharp(s)(\overline{\mathsf{n}^\sharp}[l']) \tag{30}$$

where we write $\overline{\mathsf{n}^\sharp}[l]$ (resp. $\overline{\mathsf{n}^\sharp}[l']$) for the numeric component of $NumP^\sharp$ at control point l (resp. l'), l' being the control point of statement s, and (l', l) is an arc of the program control flow.

Example 4. Consider the Java snippet in Fig. 4. From l. 4 to l. 10 is the same as in the example program of Sect. 4. Since we do not propagate the points-to graph here, the state at l. 10 is the numeric lattice $\mathsf{n}^\sharp_0$:

$$\mathsf{n}^\sharp_0 = \{\delta_{h,val} \to [-17, 41], i \to 42, max \to \top, n \to \top\} \tag{31}$$

where three scalar variables i, max and n as well as a summarized variable $\delta_{h,val}$ are involved. Note that the flow-insensitive points-to graph

$$\mathsf{p}^\sharp_{fi} = \begin{matrix}\text{tmp}\\ \text{hd}\\ \text{cur}\end{matrix} \longrightarrow \text{h} \circlearrowright_{next} \tag{32}$$

is used in the process of propagation of states but the points-to graph itself will keep unchanged (as formalized in (30)). From l. 14 to l. 21, the program finds the maximal value from the list. This value is then stored in the variable max. In case there is no positive value or the list is empty, max takes its initial value 0. We will show that at the end of the program, (l. 10):

– the scalar value max has to be in the range of $[0, 41]$

The propagation of states from lattice $\mathsf{n}^\sharp_0$ is shown in Fig. 5.

```
1  //create a list of integers
2  List hd = null, cur, tmp;
3  int i, n, max;
4  for (i = -17;i < 42;i++){
5    List tmp = new List();
   // h
6    tmp.val = i;
7    tmp.next = hd;
8    hd = tmp;
9  }
10
11 //find the maximum
12 cur = hd;
13 max = 0;
14 while (cur != null){
15   n = cur.val;
16   if (max < n){
17     max = n;
18   }
19   cur = cur.next;
20 }
21
```

Fig. 4: An example in Java. The class List has *val* and *next* as fields.

6 Experiments

We have implemented a prototype for the abstract domain $NumP^\sharp$. The implementation is called NumP. This section presents the prototype and our experimental results.

The input Java program is passed to SOOT. It computes the points-to graph and transforms the program to Jimple IR [30]. The analysis combines the abstract domains from PPL and the points-to analysis in SOOT. It infers numeric properties for each program point of the IR.

The analyzer NumP combines PPL and SOOT in a modular way. We first implement the traditional static numeric analyzer for Java. The implementation is denoted by Num, which is implemented by wrapping abstract domains in

```
1   δ → [−17, 41], i → 42, max → ⊤, n → ⊤
2   cur = hd;
3   δ → [−17, 41], i → 42, max → ⊤, n → ⊤
4   max = 0;
5     δ → [−17, 41], i → 42, max → 0, n → ⊤
6   while (hd != null){
7   δ → [−17, 41], i → 42, max → 0, n → ⊤
8   δ → [−17, 41], i → 42, max → [0, 41], n → ⊤
9       n = hd.val;
10      δ → [−17, 41], i → 42, max → 0, n → [−17, 41]
11      δ → [−17, 41], i → 42, max → [0, 41], n → [−17, 41]
12      if (max < n){
13        δ → [−17, 41], i → 42, max → 0, n → [1, 41]
14        δ → [−17, 41], i → 42, max → [0, 41], n → [1, 41]
15        max = n;
16        δ → [−17, 41], i → 42, max → [1, 41], n → [1, 41]
17    }
18    δ → [−17, 41], i → 42, max → [0, 41], n → [−17, 41]
19    hd = hd.next;
20    δ → [−17, 41], i → 42, max → [0, 41], n → [−17, 41]
21  }
22  δ → [−17, 41], i → 42, max → [0, 41], n → [0, 41]
```

Fig. 5: The propagation of states from l. 14 to l. 21 of the program in Fig. 4. The fixpoint is reached in two steps.

PPL. Num either skips unrecognized statements or conservatively approximates them using the `unconstraint` operator in PPL. The re-used components in SOOT include notably the flow-insensitive points-to analysis (from its SPARK toolkit [20]). This analyzer is denoted by Pter subsequently.

To demonstrate the effectiveness of our technique, we evaluate the analyzer on Dacapo-2006-MR2 [4] benchmark suite. The experiments were performed on a 3.06 GHz Intel Core 2 Duo with 4 GB of DDR3 RAM laptop with JDK 1.6. We tested *all* 11 benchmarks in Dacapo.

Experimental results are shown in Tab. 1 using the interval domain `Int64_Box` from PPL and the flow-insensitive points-to analysis from SOOT. The characteristics of the benchmarks are presented by the number of analyzed Jimple statements (col. 2, STATEMENT) and the number of write access statements in the form of $y_p.f_n = x_n$ or $y_p.f_n = k$ with k being a constant (col. 3, WA).

We measure PRCS (col. 4) for the number of the write access statements after which the obtained invariants are strictly more precise than Num. Q_PRCS (col. 5) is the ratio of PRCS and WA

$$\text{Q\_PRCS} \triangleq \text{PRCS/WA} \tag{33}$$

We record Q_PRCS as the metric for precision enhancement of the analyzer.

Table 1: Evaluation of NumP on the benchmark suite Dacapo-2006-MR2

Benchmark Characteristics			Precision		Time			
BENCHMARK	STATEMENT	WA	PRCS	Q_PRCS	T_NUM	T_PTER	T_NUMP	Q_T
antlr	26776	766	174	23%	00m29s	00m53s	01m36s	117%
bloat	64328	2472	943	38%	01m35s	01m02s	16m33s	632%
chart	132627	10244	3690	36%	04m17s	13m20s	83m21s	473%
eclipse	56772	820	116	14%	00m46s	00m54s	01m52s	112%
fop	198541	23482	6166	26%	03m25s	05m11s	275m28s	3203%
jython	88302	2583	1356	52%	00m57s	01m04s	05m38s	279%
hsqldb	6286	352	10	3%	00m19s	00m49s	01m16s	112%
luindex	22192	1206	250	21%	00m33s	00m54s	01m31s	105%
lusearch	26711	1503	418	28%	00m38s	00m56s	01m35s	101%
pmd	80640	3675	1316	36%	00m50s	00m55s	04m25s	252%
xalan	5197	341	3	1%	00m16s	00m49s	01m12s	111%
Mean	64397	4313	1313	25%	01m17s	02m26s	35m52s	500%

The execution time is measured for Num, Pter and NumP (col. 8, 9 and 10). The parameters T_Num and T_Pter are the times spent by Num and Pter when they analyze individually. The parameter T_NumP records the time spent our combined analysis instantiated with the interval and flow-insensitive points-to analysis.

The last column Q_T evaluates the time overhead of our analyzer. It is computed as the ratio of the time spent by our analysis to the total time spent by its component analyses.

$$\text{Q\_T} \triangleq \text{T\_NumP}/(\text{T\_Num} + \text{T\_Pter}) \tag{34}$$

The size of the analyzed Jimple statements ranges from $5,197$ (`xalan`) to $198,541$ (`fop`). The average precision metric is given in the last row of Tab. 1. The mean Q_PRCS (25%) shows a clear precision enhancement of our approach over numeric analysis only. The time overheads Q_T are generally acceptable.

In summary, we have designed an analysis in a modular way. It can be scaled to real-life programs; analyzing programs of hundreds of thousands of lines within hours can be a reasonable time budget for many applications. The precision enhancement is validated in practice.

7 Related Work

Static analysis of numeric properties has been extensively studied, especially in the framework of abstract interpretation [11]. While a large number of articles covers issues related to numeric abstractions, program analyses where both pointers and numeric values are taken into account are comparatively few.

The back-end of CodePeer[2] takes a flow-insensitive may-aliasing analysis to distinguish heap objects and to transform the analyzed programs to their SSA

[2] `http://www.adacore.com/codepeer`

forms using the global value numbering technique. The value propagation of CodePeer infers the value ranges of subtraction of variables, in other words, properties of the zone abstract domain. CodePeer goes further by taking care of inductive loop variables and the disjunctive numeric constraints, so that properties such as $b > 0 \Rightarrow a = 2 * b$ can be inferred where a or b is an inductive scalar variable. Compared with our approach, however, CodePeer uses a single zone abstract domain and do not offer the flexibility to easily plug in other abstract domains of different precision/cost tradeoffs such as the more efficient interval abstract domain or the more precise polyhedral domain. In our approach, even the capability of expressing disjunctive facts in CodePeer can be easily implemented by instantiating our numeric domain component as the powerset construction domains [2].

Efforts have been made to parametrize numeric domains with a dedicate pointer analysis. Fähndrich and Logozzo's Clousot analyzer [16] uses a value numbering algorithm to compute an under-approximation of must-alias. An optimistic assumption is then made so that Clousot regards two access paths not aliased if they do not have the same value numbering.[3] The ASTREE static analyzer [3] relies on a type based pointer analysis to deal with numeric properties of heap objects. The abstraction can be used with pointer arithmetic, union types and records of stack variables in C programs that do not have dynamic memory allocation or recursive structure. This category of static analyzers, as well as ours, can be regarded as applications of the theory of abstract domain combination which has been thoroughly studied and applied in many other contexts [28, 12, 8].

A more sophisticated heap abstraction is *shape analysis* [26]. 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 numeric abstraction of Gopan *et al.* [18] allows the integration of TVLA with existing numeric domains. The static verifier DESKCHECK [21] combines TVLA and numeric domains. It is sufficiently precise and expressive to check quantified invariants over both heap objects and numeric values. Besides the burden for users to specify the program (a problem that XISA [7, 6] 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 [25] show the mutual dependence of *conditional constant analysis* and pointer analysis. The combination is specifically designed for the conditional constant analysis and is not generalized to standard numeric domains. In particular, this approach does not directly cooperate with standard numeric

[3] This assumption is said optimistic because it is possible two access paths alias at run-time but are considered never aliased by Clousot.

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, numeric domains have been used to enhance pointer analysis. Deutsch [14] uses a parametrized numeric 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 [31] develops the structure called the *abstract fiber bundle* to formalize the idea of embedding an abstract numeric lattice within a symbolic structure. The structure enables the using of the large number of existing numeric abstractions to encode a broad spectrum of symbolic properties.

8 Conclusion

The primary objective of this work has been the automatic discovery of numeric invariants in Java-like programs, which are generally pointer-aware. We have proposed a methodology for combining numeric 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 numeric 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 numeric domains (interval domains, octagon etc.) with existing points-to analyses. The modular analyzer is able to combine advanced libraries as PPL and SPARK and it shows a clear precision enhancement with low time overhead.

Acknowledgments. The author wishes to express his gratitude to Thomas Jensen, Laurent Mauborgne and David Pichardie for their thoughtful feedback.

References

1. 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.
2. Roberto Bagnara. A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. *Sci. Comput. Program.*, 30(1-2):119–155, 1998.

3. Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.
4. 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.
5. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
6. Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.
7. Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *SAS*, pages 384–401, 2007.
8. Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Sci. Comput. Program.*, 38(1-3):27–71, 2000.
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. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
12. Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FOSSACS*, pages 456–472, 2011.
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. Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30, 2010.
17. Zhoulai Fu. *Static Analysis of Numerical Properties in the Presence of Pointers*. PhD thesis, Université de Rennes 1 – INRIA, Rennes, France, 2013.
18. Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.
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. Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, pages 71–99, 2010.

22. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004.
23. Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.
24. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
25. Anthony Pioli and Michael Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical report, IBM T. J. Watson Research Center, 1999.
26. 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.
27. A. Simon. *Value-Range Analysis of C Programs*. Springer, August 2008.
28. Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *VMCAI*, pages 375–395, 2013.
29. 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.
30. Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, July 1998.
31. Arnaud Venet. Towards the integration of symbolic and numerical static analysis. In *VSTTE*, pages 227–236, 2005.
32. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.