Targeted Update – Aggressive Memory Abstraction Beyond Common Sense and its Application on Static Numeric Analysis

Zhoulai Fu

Targeted Update — Aggressive Memory Abstraction Beyond Common Sense and its Application on Static Numeric Analysis

Zhoulai Fu *

IMDEA Software

Abstract. Summarizing techniques are widely used in the reasoning of unbounded data structures. These techniques prohibit strong update unless certain restricted safety conditions are satisfied. We find that by setting and enforcing the analysis boundaries to a limited scope of program identifiers, called *targets* in this paper, more cases of strong update can be shown sound, not with regard to the entire heap, but with regard to the targets. We have implemented the analysis for inferring numeric properties in Java programs. The experimental results show a tangible precision enhancement compared with classical approaches while preserving a high scalability.

Keywords: abstract interpretation, points-to analysis, abstract numeric domain, abstract semantics, strong update

1 Introduction

Static analysis of heap-manipulating programs has received much attention due to its fundamental role supporting a growing list of other analyses (Blanchet et al., 2003b; Chen et al., 2003; Fink et al., 2008). *Summarizing* techniques, where the heap is partitioned into finite groups,can manipulate unbounded data structures through *summarized dimensions* (Gopan et al., 2004). These techniques have many possible uses in heap analyses, such as points-to analysis (Emami et al., 1994) and TVLA (Lev-Ami and Sagiv, 2000), and also have been investigated as a basis underpinning the extension of classic numeric abstract domains to pointer-aware programs (Fu, 2013). Most of these analyses follow the *strong/weak update paradigm* (Chase et al., 1990) to model the effects of assignments on summarized dimensions. A strong update overwrites the data that may be accessed with a new value, whereas a weak update adds a new value to the summarized dimensions and preserves their old values. Strong update is desired whenever safe as it provides better precision.

* In addition to research facilities granted by IMDEA Software, this work has also received financial support from AX – L'Association des Anciens Élèves et Diplômés de l'École polytechnique at 5, rue Descartes 75005 Paris.

Applying strong update to a summarized dimension requires that it represent a single run-time memory. This requirement poses a difficulty for applying strong update as it is usually hard to know the element number represented by a summarized dimension. Efforts have been made to use sophisticated heap disambiguation techniques (Sagiv et al., 1999). While such approaches indeed help to find out more strong update circumstances, many of the proposed algorithms, such as *focus* and *blur* operations in shape analysis, are often hard to implement or come with a considerable complexity overhead.

The paper presents a new memory abstraction that makes strong update possible for summarized dimensions even if they do not necessarily represent a singleton. The approach is called *targeted update*. It extends the traditional notion of soundness in heap analysis by focusing the abstract semantics on a selected set of program identifiers, called *targets*.

Our major finding can be summarized as follows: By focusing on the targets, we are able to perform an aggressive analysis even if the traditional safety condition for strong update fails.

A motivating example Consider the assignment `y.f = 7`. Assume that the memory state before the assignment is informally represented in Fig. 1. The two access paths $x.f$ and $y.f$ are of integer type. The two gray clouds denoted by δ_1 and δ_2 represent two disjoint summarized dimensions. They initially store numeric values in the range of $[0, 5]$ and $[0, 9]$ respectively. An edge from an access path to a cloud indicates a may-access relation.

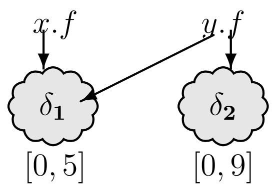

Fig. 1: Memory state before statement y.f = 7

The memory state does not tell which summarized dimension (δ_1 or δ_2) should be updated. In addition, more than one concrete memory cell may be associated with δ_1 or δ_2. Thus, traditional analysis of $y.f = 7$ performs weak update on δ_1 and δ_2. The abstract state after the assignment becomes $\delta_1 \in [0, 7] \wedge \delta_2 \in [0, 9]$, following which we infer $x.f \in [0, 7] \wedge y.f \in [0, 9]$. We call this approach *common sense strong/weak update paradigm*.

Now we present the targeted update approach. In this approach, a set of access paths needs to be selected before the analysis. The selected set is called target set. Here, if we set $\{y.f\}$ as target set, we are able to apply strong update on both δ_1 and δ_2. This is because making wrong assertions on the concrete memories of δ_1 or δ_2 that are not pointed to by $y.f$ does not contravene *the soundness with regard to the targets*: The two clouds are at most pointed to by $x.f$ and $y.f$, yet $x.f$ is not a target. The described approach is called *targeted update*. Applying targeted update with target set $\{y.f\}$ allows for precise analysis of $y.f$, but the value of the non-target $x.f$ is not tracked. The obtained $\delta_1 = 7 \wedge \delta_2 = 7$ only infers $y.f = 7$. There is no information concerning $x.f$.

It can be seen that depending on specific analysis requirement, the target set $\{y.f\}$ may not be appropriate. Imagine that we want to verify this post-condition of the statement `y.f = 7`

$$x.f \in [0,7] \wedge y.f \in [0,7] \tag{1}$$

This property cannot be verified by the strong/weak update paradigm, neither by the targeted update using $\{y.f\}$ as the target set. To use targeted update with the target set $T = \{x.f, y.f\}$ solves the problem. The summarized dimension δ_1 is now pointed to by both targets, and δ_2 by one target. Targeted update weakly updates δ_1 because updating δ_1 has an effect on both $x.f$ and $y.f$ that are targets. It strongly updates δ_2 because it is a region that can only be "observed" from $y.f$: For the concrete memories represented by δ_2 that are not pointed to by $y.f$, nothing is wrong to associate whatever values with δ_2; for the concrete memories represented by δ_2 that are truly pointed to by $y.f$, the values associated with δ_2 due to targeted update are correct. Finally, targeted update obtains $\delta_1 \in [0,7] \wedge \delta_2 = 7$, from which we infer (1).

In summary, targeted update has only responsibilities for its targets, namely, the objects pointed to by these targets, and it has no obligation to be sound with regard to the entire heap as in the common sense approach. As illustrated by the example, targeted update has two major characteristics: 1) More strong update cases on summarized dimensions can be discovered by targeted update. 2) Picking up right target set is a trade-off problem since targeted update can be very precise for targets, but it does not track non-targets.

This paper makes the following key contributions:

- We introduce the concept of *targets* and formalize the soundness notion with regard to targets (Sect. 3). The crucial insight lies in the fundamental difference of this notion of soundness with that in the common sense strong/weak paradigm.
- We derive an aggressive abstract semantics (Sect. 4 and 5) from the notion of targets. This is made possible due to a simple condition we have discovered that allows *targeted update* to be safely applied. We have formalized and proved the soundness of targeted update.
- Important design choices are discussed in Sect. 6. The implemented analyzer was tested on the SPECjvm98 benchmark suite, composed of 10 real-world Java programs.

2 Preliminaries

This section gives a brief review of some basic concepts from static program analysis that are used in this paper. Some notions of abstract interpretation are recalled In Sect. A.

General notations For a given set U, the notation $U_\perp$ represents 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 $m(x) = \perp$. We write $post[m] \in \wp(A) \to \wp(B)$ for the mapping $\lambda A_1.\{b \mid \exists a \in A_1 : m(a) = b\}$.

Syntactical notations The primary data types include scalar numbers in $\mathbb{I}$, where $\mathbb{I}$ can be integers, rationals or reals, and pointers (or references) in *Ref*. The primary syntactical entities include the universe of *local variables* and *fields*. They are denoted by *Var* and *Fld* respectively. An *access path* (Landi and Ryder, 1992) 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, Fld_τ, $Path_\tau$ and their elements with $\tau \in \{n, p\}$ to indicate their types as scalar number or reference. We use $\mathtt{Imp}_n$ to refer to the basic statements involving only numeric variables and use the meta-variables s_n to range over these statements. Similarly, we let $\mathtt{Imp}_p$ be the statements that use only pointer variables and let s_p range over these statements. Below we show the main syntactical categories and the meta-variables used in the paper.

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

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

Analysis of $\mathtt{Imp}_n$ We express a *numeric property* by a conjunction of arithmetic formulae such as $\{x + y \leq 1, x \leq 0\}$. The universe of the numeric properties is denoted by $Num^\sharp$. As usual, an *environment* maps variables to their values. We consider *numeric environments* $Num \triangleq Var_n \to \mathbb{I}_\perp$. The relationship between an environment and a property can be formalized by the relation of *valuation*. We say that $\mathsf{n} \in Num$ is a valuation of $\mathsf{n}^\sharp \in Num^\sharp$, denoted by

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

if $\mathsf{n}^\sharp$ becomes a tautology after each of its free variables is replaced by its corresponded value in n. For example, if $\mathsf{n} = \{x \to 7, y \to 7\}$, and $\mathsf{n}^\sharp = \{x + y < 15\}$ then we have $\mathsf{n} \models \mathsf{n}^\sharp$. For each statement s_n of $\mathtt{Imp}_n$, the concrete semantics is given by a standard rule of state transition $\xrightarrow{Num} (s_n) \in Num \to Num$. We write $\sqcup$ and $\triangledown$ for the join and widening operator.

In this paper, we assume that a sound abstract semantics of s_n of signature $[\![\cdot]\!]_n^\sharp \in \mathtt{Imp}_n \to (Num^\sharp \to Num^\sharp)$ is available to us. The abstract semantics is assumed to be sound with regard to the concrete $\xrightarrow{Num}$: For any n, $\mathsf{n}^\sharp$ and $s_n \in \mathtt{Imp}_n$, $\mathsf{n} \models \mathsf{n}^\sharp \Rightarrow \xrightarrow{Num} (s_n)(\mathsf{n}) \models [\![s_n]\!]_n^\sharp (\mathsf{n}^\sharp)$.

Analysis of $\texttt{Imp}_p$ A concrete state in $\texttt{Imp}_p$ is thought of as a graph-like structure representing the *environment* and *heap*. The universe of the concrete states is denoted by *Pter*. We write p to range over them.

$$\mathsf{p} \in Pter \triangleq (Var_p \to Ref_\perp) \times ((Ref \times Fld_p) \to Ref_\perp) \tag{3}$$

Points-to analysis is a dataflow analysis for detecting pointer relations. The essential process is to partition *Ref* into a finite set H and then to summarize the run-time pointer relations via elements of H and program variables. The elements of H are called *allocation sites* or *abstract references*. The process can be interfaced with a function $\triangleright$ called *naming scheme*.

$$\triangleright \in Ref \to H \tag{4}$$

In this paper, we consider a standard naming scheme that names heap objects after the control points where the objects are allocated. We assume that the naming scheme is flow-independent. That is to say, the analysis of two control branches uses the same naming scheme. Note that this is the case for points-to analysis but not for shape-analysis.

Definition 1 (Interface of traditional points-to analyzer).

$$(\texttt{Imp}_p, Pter, \xrightarrow{Pter}, Pter^\sharp, \gamma_p, [\![\cdot]\!]_p^\sharp)$$

The universe of the concrete states is denoted by Pter, and the concrete transition rule is denoted by $\xrightarrow{Pter} \in \texttt{Imp}_p \to (Pter \to Pter)$. *The universe of the abstract states is denoted by* $Pter^\sharp$. *We write* $\mathsf{p}^\sharp$ *to range over them.*

$$\mathsf{p}^\sharp \in Pter^\sharp \triangleq (Var_p \to \wp(H)) \times ((H \times Fld_p) \to \wp(H)) \tag{5}$$

Each abstract state is called a points-to graph. *The* concretization function $\gamma_p : Pter^\sharp \to \wp(Pter)$ *specifies the semantics of points-to graph. The abstract semantics* $[\![\cdot]\!]_p^\sharp$ *is assumed to be sound with regard to the concrete* $\xrightarrow{Pter}$*: For any* p, $\mathsf{p}^\sharp$ *and* $s_p \in \texttt{Imp}_p$, $\mathsf{p} \models \mathsf{p}^\sharp \Rightarrow \xrightarrow{Pter} (s_p)(\mathsf{p}) \in \gamma_p \circ [\![s_p]\!]_p^\sharp (\mathsf{p}^\sharp)$.

3 Summarizing Technique with Targets

In this section, we introduce the concept of targets and how summarizing technique with targets differs from classic summarizing technique.

The analyzed language This paper focuses on how to deal with language $\texttt{Imp}_{np}$. The statements in $\texttt{Imp}_{np}$ include these in $\texttt{Imp}_n$ and $\texttt{Imp}_p$, and 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 $\texttt{Imp}_{np}$.

$$s_{np} ::= s_n \mid s_p \mid y_p.f_n = x_n \mid x_n = y_p.f_n \tag{6}$$

We call $y_p.f_n = x_n$ or $y_p.f_n = k$ a *write access* and $x_n = y_p.f_n$ a *read access*.

A non-standard concrete semantics A concrete state in $\mathtt{Imp}_{np}$ is an environment mapping variables to values and a mapping from fields of references to values. By grouping the numeric and pointer parts, we formalize the universe of the concrete states as

$$
\begin{aligned}
State = {} & \overbrace{(Var_n \to \mathbb{I}_\perp) \times ((Ref \times Fld_n) \to \mathbb{I}_\perp)}^{Num[Var_n \cup (Ref \times Fld_n)]} && (7)\\
& \times \underbrace{(Var_p \to Ref_\perp) \times ((Ref \times Fld_p) \to Ref_\perp)}_{Pter} && (8)
\end{aligned}
$$

Thus, a state is a pair (n, p) where n can be regarded as a concrete state of $\mathtt{Imp}_n$ over $Var_n \cup (Ref \times Fld_n)$, and p as a concrete state of $\mathtt{Imp}_p$. In Sect. D, we express the concrete semantics of $\mathtt{Imp}_{np}$, denoted by $\longrightarrow^\natural$, via $\xrightarrow{Num}$ and $\xrightarrow{Pter}$.

Example 1. Consider the following program:

```
List tmp = null, hd;
int idx;
for (idx = 0; idx < 3; idx++){
  hd = new List();   // allocation site h
  hd.val = idx;
  hd.next = tmp;
  tmp = hd;
}
```

The integers $0, 1$ and 2 are stored iteratively on the heap. The head of the list is pointed to by the variable *hd*. The concrete state at the end of the program can be specified as (n, p). We write r_0, r_1 and r_2 for the concrete memories allocated at allocation site h.

$$
\begin{aligned}
\mathsf{n} &= \{(r_0, val) \to 0, (r_1, val) \to 1, (r_2, val) \to 2, idx \to 3\}\\
\mathsf{p} &= \{hd \to r_2, tmp \to r_2, (r_2, next) \to r_1, (r_1, next) \to r_0\} && (9)
\end{aligned}
$$

Common Sense Summarizing Technique A naming scheme $\rhd \in Ref \to H$ is assumed for the analysis of $\mathtt{Imp}_{np}$. In this context, the idea of summarizing technique is to use the names computed by the naming scheme to create summarized dimensions that represent the numeric values stored on the heap.

Below we show an abstraction of the concrete state (9).

$$
(\mathsf{n}^\sharp, \mathsf{p}^\sharp) = \Big(\delta_{h,val} \in [0, 2], idx = 3, \qquad hd \longrightarrow h \circlearrowleft next \Big) \tag{10}
$$

In this abstraction, the naming scheme maps the concrete r_0, r_1 and r_2 to an abstract reference $h \in H$. We can perform pointer analysis based on the naming scheme and, on the other hand, summarize numeric information on the *val* field of r_0, r_1 and r_2 by a summarized dimension related to h and *val*, denoted by $\delta_{h,val}$. The summarized dimension in this context is an element $H \times Fld_n$.

In the following, we denote $H \times Fld_n$ by Δ, and use δ to range over the pairs in Δ. We also write δ_{h,f_n} to indicate the summarized dimension corresponding to the allocation site h and the field f_n.

Definition 2. *An abstract state is defined to be a pair* $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ *of*

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

where $Num^\sharp[Var_n \cup \Delta]$ *is similar to* $Num^\sharp$*, but defined over* $Var_n \cup \Delta$*, and* $Pter^\sharp$ *is the universe of points-to graphs (Sect. 2).*

The summarizing process can be formalized through the extended naming scheme on $Ref \times Fld_n \to H \times Fld_n$, defined as $\lambda(r, f_n).(\rhd(r), f_n)$. By abuse of notation, we still write $\rhd$ for the extended naming scheme. For example, the naming scheme used in (10) satisfies $\rhd(r_i, f) = \delta_{h,f}$ for $i = 0, 1$ and 2. In (10), $\delta_{h,val} \to [0, 2]$ asserts that its concrete state (n, p) must satisfy

$$\forall (r, val) \in \rhd^{-1}(\delta_{h,val}) : \mathsf{n}(r, val) \in [0, 2] \tag{12}$$

This is common sense — a summarized dimension represents a set of concrete locations, and the fact over the summarized dimension translates to *all* the heap locations represented by the summarized dimension. Although it seems natural to require (12), we find that this kind of "contract" between the abstract and concrete states can be in some circumstances, too strong to be useful.

Assume that we have an extra statement `hd.val = 0` after l. 8. Imagine that we only want to ensure that *hd.val* becomes 0 after the statement. We cannot update $\delta_{h,val}$ to 0 because that would mean all $(r, val) \in \rhd^{-1}(\delta_{h,val})$ store the value 0, which is clearly unsound. To make a more precise analysis in this situation, we need to relax the condition (12) so that a fact over a summarized dimension does not always translate to *all* their represented concrete heap locations.

This is where *targeted update* comes in. It allows a subset $S \subseteq \rhd^{-1}(\delta_{h,val})$ in (12) to be specified so that the abstract semantics only needs to guarantee $\mathsf{n}(r, val) \in [0, 2]$ for (r, val) belonging to the specified subset S.

Targets In the context of $\mathtt{Imp}_{np}$, a *target set*, or *targets*, is a set of access paths holding numeric values on the heap. These access paths should not be local variables, and may not occur in the analyzed program syntax.

We use two operations on targets: Let t be an access path of a target set, $\mathsf{p} \in Pter$, $d \in Ref \times Fld_n$. Then $d = \mathsf{p}(t)$ reads as t *resolves to* or *points to* d under p. If p has an arc from variable x to r, then $\mathsf{p}(x.f_n) = (r, f_n)$; $\delta \in \mathsf{p}^\sharp(t)$ reads as t *resolves to* or *points to* δ under $\mathsf{p}^\sharp$. For example, in Fig. 1, we have $\mathsf{p}^\sharp(x.f) = \{\delta_1\}$ and $\mathsf{p}^\sharp(y.f) = \{\delta_1, \delta_2\}$. See Sect.B for their formal definitions.

Below we write $\mathsf{p} \in \gamma_p(\mathsf{p}^\sharp)$ to denote that p is abstracted by $\mathsf{p}^\sharp$; we write $\mathsf{n} \models [ins]\mathsf{n}^\sharp$ to denote that n is a valuation (the symbol $\models$ is introduced in Sect. 2) of $\mathsf{n}^\sharp$ with its variables substituted following *ins*. For example, let $ins = \{\delta_1 \to d_1, \delta_2 \to d_2\}$ and $\mathsf{n}^\sharp = \{\delta_1 + \delta_2 > 0, \delta > 10\}$. Then we have $[ins]\mathsf{n}^\sharp = \{d_1 + d_2 > 0, d_2 > 10\}$.

If a target set is selected and the soundness is enforced with regard to the targets, the abstract state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ represents all concrete states (n, p) as long as p is abstracted by $\mathsf{p}^\sharp$ and n can be abstracted by whatever $\mathsf{n}^{\sharp\prime}$ that is $\mathsf{n}^\sharp$ with its summarizing dimensions $\delta_1, \ldots, \delta_m$ instantiated with some $d_1, \ldots, d_m$

satisfying: For $1 \le i \le m$, $\triangleright(d_i) = \delta_i$ and d_i can be reached by targets, *i.e.*, $\exists t \in T : d_i = \mathsf{p}(t)$.

Definition 3. *Let T be the target set. The concretization of a state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \in NumP^\sharp$ is defined as*

$$\gamma_{\langle T \rangle}(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \{(\mathsf{n}, \mathsf{p}) \mid \mathsf{p} \in \gamma_p(\mathsf{p}^\sharp), \forall ins \in Ins_{\mathsf{p}}\langle T \rangle : \mathsf{n} \models [ins](\mathsf{n}^\sharp)\} \quad (13)$$

with $Ins_{\mathsf{p}}\langle T \rangle \triangleq \{ins \in \Delta \to D \mid \forall (\delta, d) \in ins : \triangleright(d) = \delta \ \wedge \ d \in post[\mathsf{p}](T)\}$. Read it as, an element (n, p) is in the concretization $\gamma_{\langle T \rangle}(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$, if p is in the concretization of $\mathsf{p}^\sharp$, and n is in the concretization of $[ins]\mathsf{n}^\sharp$ where ins is called an instantiation *mapping summarized dimensions to concrete $d \in D$ that are pointed to by the targets T.*

Below, we present the abstract semantics of statements in $\texttt{Imp}_{np}$, called *targeted update.*

4 Targeted Update — the case of write access $y_p.f_n = x_n$

Algorithm Targeted update uses two operators: The *local strong update* operator $[\![\delta = x_n]\!]^S$ assigns x_n to δ, regarding x_n and δ as scalar variables. For example, if it is interval domain on which targeted update is built, we have

$$[\![\delta = x_n]\!]^S (\{\delta \in [1, 2], x_n \in [3, 4]\}) = \{\delta \in [3, 4], x_n \in [3, 4]\} \quad (14)$$

Another operator $[\![\delta = x_n]\!]^W$ is called *local weak update* operator. It assigns x_n to δ and then joins the result with its original state, for example,

$$[\![\delta = x_n]\!]^W (\{\delta \in [1, 2], x_n \in [3, 4]\}) = \{\delta \in [1, 4], x_n \in [3, 4]\} \quad (15)$$

It is clear that both operators can be computed from traditional numeric domains.

The input of targeted update is an abstract state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \in NumP^\sharp$ and a pre-selected target set T. We do not care about how this set is selected for now. Targeted update first computes the summarized dimensions to which $y_p.f_n$ resolves, namely $\mathsf{p}^\sharp(y_p.f_n)$. Each summarized dimension δ is then treated one by one.[1] If the following condition holds:

$$\delta \text{ is pointed to by no target in } T \backslash \{y_p.f_n\} \quad (TU)$$

then local strong update will be performed on δ; otherwise, local weak update has to be performed on δ. The above condition is referred to as (TU) condition subsequently. This algorithm for the abstract semantics is presented in Algo. 1.

[1] Dealing with δ in different orders could have an influence on precision, but this point is not studied in the paper.

Algorithm 1: TARGETED UPDATE FOR $y_p.f_n = x_n$

Input: Abstract state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$, targets T
Output: The abstract state after targeted update $[\![y_p.f_n = x_n]\!]^\sharp_{\langle T \rangle} (\mathsf{n}^\sharp, \mathsf{p}^\sharp)$

$\mathsf{n}^\sharp{\prime} \leftarrow \mathsf{n}^\sharp$
for $\delta \in \mathsf{p}^\sharp(y_p.f_n)$ **do**
  if *there exists no* $t \in T \backslash \{y_p.f_n\}$ *satisfying* $\delta \in \mathsf{p}^\sharp(t)$ **then**
    $\mathsf{n}^\sharp{\prime} \leftarrow [\![\delta = x_n]\!]^S (\mathsf{n}^\sharp{\prime})$
  else
    $\mathsf{n}^\sharp{\prime} \leftarrow [\![\delta = x_n]\!]^W (\mathsf{n}^\sharp{\prime})$
  end if
end for
return $\mathsf{n}^\sharp{\prime}, \mathsf{p}^\sharp$

Remark 1. Automatically finding targets adapted to specific problem requirements is a problem in itself. In our implementation, we use the *numeric access paths* (excluding scalar variables) that appear syntactically in the program as targets.

Comparison with Strong/Weak Update Below, we present a case study. It shows how targeted update works and in which way it differs from the common sense strong/weak update paradigm.

Example 2. Assume that a program has three numeric access paths: t, $y_p.f_n$ and s, and there are three summarized dimensions: δ_1, δ_2 and δ_3. Assume that the access paths resolve to summarized dimensions as depicted:

$$\begin{tikzcd} t \arrow[drr] & y_p.f_n \arrow[dl] \arrow[d] \arrow[dr] & s \arrow[dl] \arrow[d] \\ \delta_1 & \delta_2 & \delta_3 \end{tikzcd} \tag{16}$$

namely, $\mathsf{p}^\sharp(t) = \{\delta_3\}$, $\mathsf{p}^\sharp(y_p.f_n) = \{\delta_1, \delta_2, \delta_3\}$, $\mathsf{p}^\sharp(s) = \{\delta_2, \delta_3\}$. We shall compare targeted update and strong/weak update paradigm of $y_p.f_n = x_n$.

The concrete semantics of $y_p.f_n = x_n$ is known: It modifies one element of $d \in \rhd^{-1}(\delta_1) \cup \rhd^{-1}(\delta_2) \cup \rhd^{-1}(\delta_3)$. It is clear that the information from (16) does not help to identify the one among δ_1, δ_2, and δ_3 that will be modified by the statement. In addition, this specific δ may have more than one concrete represented element. Thus, the traditional approach performs weak update which amounts to a conservative join of $[\![\delta_1 = x_n]\!]^W (\mathsf{n}^\sharp)$, $[\![\delta_2 = x_n]\!]^W (\mathsf{n}^\sharp)$ and $[\![\delta_3 = x_n]\!]^W (\mathsf{n}^\sharp)$. Formally, the weak update is defined as

$$[\![y_p.f_n = x_n]\!]^\sharp (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \left(\sqcup_{\delta \in \mathsf{p}^\sharp(y_p.f_n)} [\![\delta = x_n]\!]^W (\mathsf{n}^\sharp) \right), \mathsf{p}^\sharp \tag{17}$$

Now, let us consider targeted update. Assume that all three access paths are targets, $T = \{t, y_p.f_n, s\}$. Because only δ_1 satisfies (TU) condition, targeted update abstracts $y_p.f_n = x_n$ as a composition of local weak update of δ_2 and δ_3, and local strong update of δ_1, namely, $[\![\delta_3 = x_n]\!]^W \circ [\![\delta_2 = x_n]\!]^W \circ [\![\delta_1 = x_n]\!]^S$. Formally, we define targeted update as follows.

Definition 4. *Let T be a set of targets, $(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \in NumP^\sharp$. Define the targeted update for $y_p.f_n = x_n$:*

$$[\![y_p.f_n = x_n]\!]^\sharp_{\langle T\rangle} (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq [\![\delta_1 = x_n]\!]^{\eta(\delta_1)} \circ \cdots \circ [\![\delta_M = x_n]\!]^{\eta(\delta_M)}\, \mathsf{n}^\sharp, \mathsf{p}^\sharp \tag{18}$$

with $\{\delta_1, \ldots, \delta_M\} = \mathsf{p}^\sharp(y_p.f_n)$,

$$\eta \triangleq \lambda\delta : \mathsf{p}^\sharp(y_p.f_n). \begin{cases} S & \textit{if } \{t \in T \mid t \neq y_p.f_n \wedge \delta \in \mathsf{p}^\sharp(t)\} = \emptyset \\ W & \textit{otherwise} \end{cases} \tag{19}$$

Correctness The correctness of the abstract semantics can be formalized as follows.

Theorem 1. *Let T be a target set. For any abstract state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ of $NumP^\sharp$ and any $(\mathsf{n}, \mathsf{p}) \in \gamma_{\langle T\rangle}(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$. We have*

$$\longrightarrow^\natural(y_p.f_n = x_n)(\mathsf{n}, \mathsf{p}) \in \gamma_{\langle T\rangle} \circ [\![y_p.f_n = x_n]\!]^\sharp_{\langle T\rangle} (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \tag{20}$$

We need a lemma for the proof. If the (TU) condition holds, the summarized dimension δ specified in the condition is pointed to by at most one target. Observationally, δ is a singleton representing only one object, although δ may represent more than one object that is not necessarily pointed to by targets.

This intuition is formalized as the lemma below. We write $tu(T, \mathsf{p}^\sharp, y_p.f_n, \delta)$ as a shortcut for (TU), namely $\nexists t \in T\backslash\{y_p.f_n\} : \delta \in \mathsf{p}^\sharp(y_p.f_n)$. The proof of the lemma needs a property as stated of points-to graph: For any concrete p and abstract $\mathsf{p}^\sharp$ such that $\mathsf{p} \in \gamma_p(\mathsf{p}^\sharp)$, if access path $\boldsymbol{u}$ resolves to $d \in Ref \times Fld_n$, i.e. $\mathsf{p}(\boldsymbol{u}) = d$, then we have $\rhd(d) \in \mathsf{p}^\sharp(\boldsymbol{u})$. This property ensures, for example, if $\mathsf{p}(x) = r$ in the concrete, then $\mathsf{p}^\sharp(x)$ has to contain $\rhd(r)$.

Lemma 1. *Assume that $tu(T, \mathsf{p}^\sharp, y_p.f_n, \delta)$ holds. Then, for any $\mathsf{p} \in \gamma_p(\mathsf{p}^\sharp)$ and $ins \in Ins_{\mathsf{p}}\langle T\rangle$, we have $ins(\delta) = \mathsf{p}(y_p.f_n)$.*

Proof (Proof of Lem. 1). Because $ins \in Ins_{\mathsf{p}}\langle T\rangle$, we have $ins(\delta)$ must be pointed to by targets in T.

$$ins(\delta) \in \{\mathsf{p}(t) \mid t \in T, t \neq y_p.f_n\} \cup \{\mathsf{p}(y_p.f_n)\} \tag{21}$$

Condition (TU) combined with the semantics of points-to graph tells that the first part of (21) has to be empty. Otherwise, we have some $t \in T\backslash\{y_p.f_n\}$ pointing to δ, which contradicts $tu(T, \mathsf{p}^\sharp, y_p.f_n, \delta)$. By consequence, we have $ins(\delta) = \mathsf{p}(y_p.f_n)$. □

This lemma plays a crucial role in proving the correctness of the abstract semantics. We give its proof sketch in Sect. F.

5 Targeted Update — the case of read access $x_n = y_p.f_n$, s_n and s_p

We have developed an abstract semantics for the write access statement using the soundness notion with regard to targets. This section presents our abstract semantics for other types of statements in $\texttt{Imp}_{np}$.

Case for $x_n = y_p.f_n$ Assume that $y_p.f_n$ only resolves to δ. It is tempting, but wrong, to abstract statement as in traditional numeric analysis, *i.e.*, $[\![x_n = \delta]\!]^\sharp_n$. Consider `a = x.f; b = y.f; if (a < b){...}`. Assume that $\mathsf{p}^\sharp(x.f) = \mathsf{p}^\sharp(y.f) = \{\delta\}$. If the abstract semantics relates a (resp. b) with δ after $a = x.f$ (resp. $b = y.f$), the analysis will wrongly argue that the following `if` branch can never be reached. The above reasoning is wrong because we should not, in general, correlate a summarized dimension with a scalar variable.

Gopan *et al.* have pointed out that to assign a summarized dimension δ to a non-summarized dimension x_n takes three steps: First, *extend* δ to a fresh dimension δ' (using the operator $\mathrm{expand}^\sharp_{\delta,\delta'}$ that copies dimensions. Then, *relate* x_n with δ' using traditional abstract semantics for assignment $[\![x_n = \delta']\!]^\sharp_n$. Finally, the newly introduced dimension δ' has to be *dropped* (using the operator $\mathrm{drop}^\sharp_{\delta'}$ that removes dimensions). See (Gopan et al., 2004) for the details of $\mathrm{drop}^\sharp_{\delta'}$ and $\mathrm{expand}^\sharp_{\delta,\delta'}$.

In summary, Gopan's operator *copies* the values of the summarized dimension to the scalar variable but keeps them uncorrelated. The following operator is used to assign a summarized dimension δ to a scalar variable x_n.

$$G(x_n, \delta) \triangleq \lambda \mathsf{n}^\sharp.\, \mathrm{drop}^\sharp_{\delta'} \circ [\![x_n = \delta']\!]^\sharp_n \circ \mathrm{expand}^\sharp_{\delta,\delta'}\, \mathsf{n}^\sharp \tag{22}$$

For example, the property $G(x_n = \delta)\{\delta > 1\} = \{x_n > 1, \delta > 1\}$ after applying $x = \delta$. We see that scalar variable x_n and summarized dimension δ cannot be related, even if the underlined numeric domain is relational.

Remark 2. The lack of correlation between δ and x_n reveals another source of imprecision of the classic soundness notion, besides its weak update semantics.

Sharper analysis can be obtained thanks to the notion of targets. In Lem. 1, we have shown an important consequence of (TU), that is, the underlined summarized dimension δ represents a single concrete object among the objects pointed to by the targets. This lemma allows us to deal with δ satisfying (TU) as a scalar variable.

Consider the read access $x_n = y_p.f_n$. Let $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ be the input abstract state, T be the targets. If $y_p.f_n \notin T$, we have to unconstrain x_n. If $y_p.f_n \in T$ and $\mathsf{p}^\sharp(y_p.f_n) = \{\delta_1, \ldots, \delta_M\}$, targeted update joins the effects of assigning δ_i to x_n for $1 \le i \le M$. For each δ_i, if (TU) satisfies, the effect of assigning δ_i to x_n is the same as $[\![x_n = \delta_i]\!]^\sharp_n(\mathsf{n}^\sharp)$, as if δ_i is a scalar variable; if (TU) fails, the best we can do is to copy the possible values of δ_i into x_n, which amounts to using Gopan's operator (22). This is summarized in Algo. 2. That is,

$$\llbracket x_n = y_p.f_n \rrbracket^\sharp_{\langle T \rangle} (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \begin{cases} \llbracket x_n = ? \rrbracket^\sharp_n \mathsf{n}^\sharp, \mathsf{p}^\sharp & y_p.f_n \notin T \\ \bigsqcup_{\delta \in \mathsf{p}^\sharp(y_p.f_n)} \llbracket x_n = \delta \rrbracket^{\eta(\delta)} \mathsf{n}^\sharp, \mathsf{p}^\sharp & y_p.f_n \in T \end{cases} \tag{23}$$

where the operator $\llbracket x_n = ? \rrbracket^\sharp_n$ unconstrains x_n, η is the shortcut defined in (19), and

$$\llbracket x_n = \delta \rrbracket^S \triangleq \llbracket x_n = \delta \rrbracket^\sharp_n , \qquad \llbracket x_n = \delta \rrbracket^W \triangleq G(x_n, \delta) \tag{24}$$

Algorithm 2: TARGETED UPDATE FOR $x_n = y_p.f_n$

Input: Abstract state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$, targets T
Output: The abstract state after targeted update $\llbracket x_n = y_p.f_n \rrbracket^\sharp_{\langle T \rangle} (\mathsf{n}^\sharp, \mathsf{p}^\sharp)$

if $y_p.f_n \notin T$ **then**
  return $\llbracket x_n = ? \rrbracket^\sharp_n (\mathsf{n}^\sharp), \mathsf{p}^\sharp$
$\mathsf{n}^\sharp\prime \leftarrow \bot$
for $\delta \in \mathsf{p}^\sharp(y_p.f_n)$ **do**
  if *there exists no* $t \in T \backslash \{y_p.f_n\}$ *satisfying* $\delta \in \mathsf{p}^\sharp(t)$ **then**
    $\mathsf{n}^\sharp\prime \leftarrow \mathsf{n}^\sharp\prime \sqcup \llbracket x_n = \delta \rrbracket^\sharp_n (\mathsf{n}^\sharp\prime)$
  else
    $\mathsf{n}^\sharp\prime \leftarrow \mathsf{n}^\sharp\prime \sqcup G(x_n, \delta)$
  end if
end for
return $\mathsf{n}^\sharp\prime, \mathsf{p}^\sharp$

Case for s_n If s_n is an assignment in $\mathtt{Imp}_n$, it can be treated in the same way as in traditional numeric analysis using its abstract transfer function $\llbracket \cdot \rrbracket^\sharp_n$ (Sect. 2). In this paper, the transfer function for updating $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ with s_n is defined as:

$$\llbracket s_n \rrbracket^\sharp_{\langle T \rangle} (\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq (\llbracket s_n \rrbracket^\sharp_n \mathsf{n}^\sharp, \mathsf{p}^\sharp) \tag{25}$$

Case for s_p Targeted update tracks the heap objects pointed to by the targets. An important thing to note is that s_p may cause changes to what objects the access paths are pointing—necessitating changes to the numeric portion of the abstract state. Subsequently, we write s_p in the form of 'l=r'.

Given a target set T and an abstract state $(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \in NumP^\sharp$. Taking an arbitrary $(\mathsf{n}, \mathsf{p}) \in \gamma_{\langle T \rangle}(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$, we want to find $\mathsf{n}^\sharp\prime$ so that $(\mathsf{n}, \xrightarrow{Pter} (s_p)\mathsf{p})$ is in the concretization of $(\mathsf{n}^\sharp\prime, \llbracket s_p \rrbracket^\sharp_p (\mathsf{p}^\sharp))$. The hypothesis $(\mathsf{n}, \mathsf{p}) \in \gamma_{\langle T \rangle}(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ states that $\mathsf{n} \models [ins]\mathsf{n}^\sharp$ for any $ins \in Ins_{\mathsf{p}}\langle T\rangle$; for the sake of soundness, the updated $\mathsf{n}^\sharp{}'$ has to satisfy $\mathsf{n} \models [ins]\mathsf{n}^\sharp{}'$ for any $ins \in Ins_{\xrightarrow{Pter}(s_p)\mathsf{p}}\langle T\rangle$. Following Def. 3, it suffices to unconstrain all summarized dimensions of $\mathsf{n}^\sharp{}'$ in the form of $\rhd(d)$ with $d \in post[\xrightarrow{Pter}(s_p)\mathsf{p}](T)\backslash post[\mathsf{p}](T)$. Let $M \triangleq post[\xrightarrow{Pter}(s_p)\mathsf{p}](T) \cap post[\mathsf{p}](T)$. We can show that $M \supseteq \{\mathsf{p}(t) \mid t \in T, t$ does not have l as prefix$\}$. This is because for any $\mathsf{p}(t)$ such that $t \in T$ and t does not have l as prefix, $\mathsf{p}(t) \in post[\mathsf{p}](T)$ immediately implies $\mathsf{p}(t) \in post[\xrightarrow{Pter}(s_p)](T)$.

In conclusion, a conservative way to model s_p is to unconstrain targets that do not necessarily point to where they previously pointed. Thus, we unconstrain all $\mathsf{p}^\sharp(t)$ such that $t \in T$ and t has l as prefix. For example, in `x = new;` we unconstrain δ if it is pointed to by the target $x.val$. The transfer function for s_p is modeled as:

$$\llbracket s_p \rrbracket^\sharp_{\langle T\rangle}(\mathsf{n}^\sharp, \mathsf{p}^\sharp) \triangleq \bigsqcup_{\delta \in uncons_{\langle T\rangle}(s_p, \mathsf{p}^\sharp)} \llbracket \delta = ?\rrbracket^\sharp_n \mathsf{n}^\sharp, \quad \llbracket s_p \rrbracket^\sharp_p \mathsf{p}^\sharp \tag{26}$$

Here, $uncons_{\langle T\rangle}(s_p, \mathsf{p}^\sharp) \triangleq \{\delta \mid s_p = \text{‘}l{=}r\text{’},\ \exists t \in T : t \text{ has } l \text{ as prefix} \wedge \delta \in \mathsf{p}^\sharp(t)\}$.

6 A Discussion of Some Important Design Choices

Targets Our implementation uses the numeric access paths excluding variables that appear syntactically in the program as targets. Without prior knowledge of specific program properties to be verified, this design choice seems to give a trade-off between expressiveness and precision. Although this target set may appear large, our experiments (Sect. 8) show that targeted update using this target set still provides a significant precision enhancement while covering common cases where program properties to be expressed only use program syntax.

Join and widening The design of the join operator is usually a difficult step for developing abstract domains. We have assumed (Sect. 2) that the naming scheme should be flow independent. Thanks to the naming scheme hypothesis, our join operator seems to be delightfully uncomplicated: We just compute the join (or widening) component-wise. Then, if a concrete state (n, p) is in $\gamma_{\langle T\rangle}(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp_1)$ or in $\gamma_{\langle T\rangle}(\mathsf{n}^\sharp_2, \mathsf{p}^\sharp_2)$, it is also in the concretization of $(\mathsf{n}^\sharp_1 \sqcup \mathsf{n}^\sharp_2, \mathsf{p}^\sharp_1 \cup \mathsf{p}^\sharp_2)$. The case for widening is similar.

$$(\mathsf{n}^\sharp_1, \mathsf{p}^\sharp_1) \sqcup^\sharp (\mathsf{n}^\sharp_2, \mathsf{p}^\sharp_2) = (\mathsf{n}^\sharp_1 \sqcup \mathsf{n}^\sharp_2, \mathsf{p}^\sharp_1 \cup \mathsf{p}^\sharp_2) \tag{27}$$

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

Constraint system with a flow-insensitive points-to analysis As in the implementation of (Fu, 2014), we use a flow-insensitive points-to analysis to simplify the states propagation. The analysis is done in a pre-analysis phase and does not participate with the propagation of numeric lattices. The obtained flow-insensitive points-to graph is then used at each control point as a superset of the flow-sensitive points-to graph.

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. Using the single flow-insensitive points-to graph for all program control points can be modeled as an analysis that is initialized with an over-approximation of the least fixpoint of a flow-sensitive analysis that propagates in the style of `skip`.

Let $F^\sharp(s) \triangleq \lambda \mathsf{n}^\sharp . \mathit{fst} \circ [\![s]\!]^\sharp_{\langle T \rangle} (\mathsf{n}^\sharp, \mathsf{p}^\sharp_{fi})$, 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 the constraint system that operates on numeric lattice $\mathsf{n}^\sharp$ only (rather than on $(\mathsf{n}^\sharp, \mathsf{p}^\sharp)$ pair):

$$\overline{\mathsf{n}^\sharp}[l] \sqsupseteq F^\sharp(s)(\overline{\mathsf{n}^\sharp}[l']) \tag{29}$$

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' is the control point of statement s, and (l', l) is an arc in the program control flow.

Intra-procedural numeric analysis While the points-to graph is computed by an interprocedural pointer analysis, the static numeric anlaysis is intentionally left intra-procedural.

Existing numeric domains, in particular the relational ones, are generally sensitive to the size of the program and number of variables. The objective of scalability is hard to achieve if the problem solving has to iterate through all the program call-graph. To take variables in all the procedures as a whole necessarily incurs a high complexity for the numeric part in our analysis. To give an idea of this complexity, our experiments on the abstract domains in PPL show that octagonal analysis can hardly run on several hundreds of variables, and polyhedral analysis can quickly time out with more than 30 variables; on the other hand, a real-world Java program, with all its procedures put in together, could easily reach tens of thousands of variables to be analyzed.

A known workaround exists. The pre-analysis of *variable packing* technique allows ASTREE (Blanchet et al., 2003a) to successfully scale up to large sized C programs. We regard intra-procedural numeric analysis as a lightweight alternative to variable packing: Variables are related only if they are in the same procedure. In this way, we do not need to invent strategies to pack variables.

7 An Example

We discuss a Java program with interesting operations on a single linked list. Fig. 2 presents the program. Here, our goal is to show how targeted update works in practice and to prove two properties that are challenging for a human. The analysis results from our implemented analyzer are shown in Sect. E.

```
1  List hd, node;  int idx;
2  hd = new List(); //allocation site h1
3  hd.val = 0;
4  hd.next = null;
5  for (idx = -17; idx < 42; idx++){
6       node = new List(); //allocation site h2
7       node.val = idx;
8       node.next = hd.next;
9       hd.next = node;
10      hd.val = hd.val + 1;
11 }
12 return;
```

Fig. 2: A Java program

Example 3. Observe that there are two allocation sites h_1 and h_2 in the program, with the head of the list stored in h_1 and the body of the list stored in h_2. The head node has a special meaning. It is used to indicate of length of the list. From l. 1 to l. 4, the program creates an empty list with a single head node. From l. 5 to l. 11, a list of integers is iteratively stored on the list. Within the loop, the head node is updated (l. 10) to track list length whenever a new list cell is created.

Targeted update, instantiated with polyhedral analysis, is able to infer the following properties:

- *Prop1:* At the loop entry (l. 5), $hd.val \in [0, 60] \wedge hd.val - idx = 17$.
- *Prop2:* From l. 5 to l. 10, $hd.val - node.val = 17$.
- *Prop3:* At the exit of the loop (l. 12), $hd.val = 60$.

Targeted update works as follows: First, it pre-analyzes the program with flow-insensitive points-to analysis.

$$\begin{tikzcd} hd \arrow[dr] & & node \arrow[d] \\ & h_1 \arrow[r, bend left] & h_2 \arrow[loop right, "next"] \end{tikzcd} \tag{30}$$

All numeric access paths appeared in program syntax that are not variables are taken as targets: $T = \{hd.val, node.val\}$. By computing $\{\delta \mid \exists t \in T, \delta \in \mathsf{p}^{\sharp}_{fi}(t)\}$, targeted update obtains two summarized dimensions $\delta_{h_1,val}$ and $\delta_{h_2,val}$. The initial abstract state is set to $\{\delta_{h_1,val} \to \top, \delta_{h_2,val} \to \top, idx \to \top\}$. Then, we apply transfer functions of targeted update and solve the constraint system (29). For example, the statements at l. 3 and l. 7 are treated as write access $y_p.f_n = x_n$. The statement at l .10 is transformed by SOOT into three short ones: `tmp1 = hd.val`, `tmp2 = tmp1 + 1` and `hd.val = tmp2`. They are treated as read access, s_n and write access statements, respectively. Finally, targeted update obtains (1) at l. 5: $\delta_{h_1.val} \in [0, 60] \wedge \delta_{h_1,val} - idx = 17$, (2) at l. 5 to l. 10: $\delta_{h_1.val} - \delta_{h_2.val} = 17$ and (3) At l. 12: $\delta_{h_1.val} = 60$. From these, we deduce *Prop1*, *Prop2* and *Prop3* respectively (based on the concretization function defined in Def. 3).

These properties are interesting and useful. *Prop1* tells a non-trivial loop invariant involving access paths and scalar variables. *Prop2* is particularly difficult to infer: *hd.val* and *node.val* have an invariant difference 17 because this is the case at the loop entry; in addition, *node.val* increments by one (because it is correlated with the *idx* at l. 7) at each iteration, and *hd.val* increments by one as well (l. 10). *Prop3* gives a precise value stored in the head node, indicating that the list length is tracked as 60, precisely.

Remark 3. Targeted update is able to infer these relations because the summarized dimensions $\delta_{h_1,val}$ and $\delta_{h_2,val}$ lose their original sense: They can be correlated with scalar variables and strongly updated because (TU) condition is satisfied there. In addition, since targeted update is built on traditional numeric domains, we can take the best from these, such as the very precise polyhedral abstraction and the widening/narrowing techniques (Cousot and Cousot, 1992) used in this example.

8 Experiments

The implemented targeted update is built on the static numeric analyzer `NumP` developed in (Fu, 2014). Our implementation of targeted update is called `T-NumP`. The analyzed language of `T-NumP` is Jimple (Vallée-Rai et al., 1999). The compiler framework SOOT is used as the analysis front-end. It offers a range of pointer analyses as well, including the points-to analysis and the side-effect analysis (to approximate the effects of invocation). The default flow-insensitive points-to analysis used in SOOT is denoted by `Pter` subsequently. For the purpose of comparison, we have implemented a traditional static numeric analyzer for Java by wrapping abstract domains in PPL. The implemented analyzer is called `Num`.

Assessment To demonstrate the effectiveness of our technique, we evaluate it on the SPECjvm98 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 the 10 benchmarks in SPECjvm98. The corresponding results are given in Tab. 1 and 2. The characteristics of the benchmarks are presented by the number of the analyzed Jimple statements (col. 2, STATEMENT), the number of write access statements in the form of $y_p.f_n = x_n$ or $y_p.f_n = k$ (col. 3, WA), and the number of read access statements in the form of $x_n = y_p.f_n$ (col. 4, RA). Experimental results are shown in Tab. 1 where we use the interval domain `Int64_Box` of PPL.

Three parameters TU, PRCS, and SCALAR (col. 5-7) are measured to estimate the precision gain. The parameter TU counts the number of write access statements before which condition (TU) is satisfied. We record PRCS for the

Table 1: Evaluation of targeted update on the benchmark suite SPECjvm98: Interval + Spark

Benchmark Characteristics				Precision			Time			Metrics			
BENCHMARK	STATEMENT	WA	RA	TU	PRCS	SCALAR	T_NUM	T_PTER	T_TNUMP	Q_TU	Q_PRCS	Q_SCALAR	Q_T
_200_check	2307	25	48	19	18	6	00m12s	02m36s	03m13s	76%	72%	13%	115%
_201_compress	2724	96	142	89	55	9	00m07s	02m39s	03m34s	93%	57%	6%	129%
_202_jess	12834	232	646	212	102	2	00m16s	02m43s	05m02s	91%	44%	0%	169%
_205_raytrace	5465	53	64	52	24	0	00m05s	02m35s	03m35s	98%	45%	0%	134%
_209_db	2770	32	65	31	19	0	00m04s	02m41s	03m47s	97%	59%	0%	138%
_213_javac	25973	342	1362	312	143	25	00m12s	04m15s	10m12s	91%	42%	2%	229%
_222_mpegaudio	14604	138	247	124	62	6	00m18s	02m50s	04m15s	90%	45%	2%	136%
_227_mtrt	5466	53	64	52	24	0	00m06s	02m40s	03m42s	98%	45%	0%	134%
_228_jack	12221	462	414	436	102	7	00m31s	02m45s	06m03s	94%	22%	2%	185%
_999_checkit	3038	38	53	29	19	0	00m05s	02m38s	03m44s	76%	50%	0%	137%
Mean										90%	48%	3%	151%

number of the write access statements after which the obtained invariants are strictly more precise than `Num`. Improvement on scalar variables is assessed by the number of read-access statements after which the obtained numeric invariant by `T-NumP` is strictly more precise than `Num` in terms of scalar variables (summarized dimensions are unconstrained for this comparison). The execution time is measured for `Num`, `Pter`, and `T-NumP` (col. 8-10). The parameters T_Num and T_Pter are the times spent by `Num` and `Pter` when they analyze individually. The parameter T_TNUMP records the time of our analysis.

The last four columns compute the metrics for assessment. The metrics Q_TU $\triangleq$ TU/WA and Q_PRCS $\triangleq$ PRCS/WA (col. 11-12) are the ratios of TU and PRCS to the number of write access statements. The metrics Q_SCALAR $\triangleq$ SCALAR/RA (col. 13) is defined with regard to read-access statements. The metric Q_T $\triangleq$ T_TNUMP/(T_Num+T_Pter) (col. 14) records the ratio of the time spent by our analysis to the total time of its component analyses.

The size of the analyzed Jimple statements ranges from 2307 (`_200_check`) to 25973 (`_213_javac`).[2] We observe that T_Pter is always much larger than T_Num. This is because the points-to analysis is interprocedural while the numeric analysis is run procedures by procedures. Our analysis relies on the pointer analysis and is thus bottlenecked by it in terms of efficiency. Still, the time spent for the benchmark takes several minutes, with an average Q_T = 151%. The average precision metrics is calculated on the last row of Tab. 1. Q_TU = 90%, Q_PRCS = 48% show a clear precision enhancement of our approach over traditional approaches.

Please mind the gap between TU and PRCS in Tab. 1 (and between Q_TU nd Q_PRCS as well). Besides the non-monotonicity of widening operators (Cortesi and Zanioli, 2011), we observe that the practical reason causing this disparity is that targeted update, in the context of non-relational analysis (as the interval analysis above), is helpless in dealing with write-access statements in the form of $y_p.f_n = x_n$ as long as no information on x_n has been gathered.

[2] The Jimple statements are generally less than in the source program, because SOOT typically analyzes a subset of its call-graph nodes.

This point can be remedied by relational analysis. Tab. 2 shows our experimental results with octagonal analysis and the same points-to analysis as above. Since the condition (TU) can not be influenced by numeric analysis, we obtain the same Q_TU as in Tab. 1. The parameters Q_PRCS and Q_SCALAR can be greatly improved due to the relational analysis, with similar time overhead Q_TU as in Tab. 1.

Table 2: Evaluation of targeted update on the benchmark suite SPECjvm98: Octagonal + Spark

Benchmark Characteristics				Precision			Time			Metrics			
BENCHMARK	STATEMENT	WA	RA	TU	PRCS	SCALAR	T_NUM	T_PTER	T_TNUMP	Q_TU	Q_PRCS	Q_SCALAR	Q_T
_200_check	2307	25	48	19	19	6	00m13s	02m44s	03m48s	76%	76%	13%	129%
_201_compress	2724	96	142	89	93	70	00m09s	03m18s	05m16s	93%	97%	49%	153%
_202_jess	12834	232	646	212	215	52	00m36s	02m46s	06m38s	91%	93%	8%	197%
_205_raytrace	5465	53	64	52	52	8	00m10s	02m38s	03m52s	98%	98%	13%	138%
_209_db	2770	32	65	31	31	13	00m08s	02m42s	03m51s	97%	97%	20%	136%
_213_javac	25973	342	1362	312	244	156	02m35s	05m31s	14m28s	91%	71%	11%	179%
_222_mpegaudio	14604	138	247	124	117	36	00m39s	02m45s	06m44s	90%	85%	15%	198%
_227_mtrt	5466	53	64	52	52	8	00m21s	02m37s	03m58s	98%	98%	13%	134%
_228_jack	12221	462	414	436	410	168	00m34s	02m43s	08m06s	94%	89%	41%	247%
_999_checkit	3038	38	53	29	28	6	00m09s	02m52s	04m46s	76%	74%	11%	158%
Mean										90%	88%	19%	167%

The experimental results show that targeted update discovers significantly more program properties in summarized dimensions and scalar variables as well, at a cost comparable to that of running the numeric and pointer analysis separately.

9 Related Work

This research continues the work in (Fu, 2014) that addresses the general issue of lifting numeric domains to heap-manipulating programs.

Memory abstraction using strong and weak updates (Chase et al., 1990; Wilson and Lam, 1995) is common sense. Efforts have been made to enable safe application of strong update. Sagiv *et al.* used the *focus* operation (that isolates individual elements of the summarized dimensions) of shape analysis (Sagiv et al., 1999) to apply strong update. Fink *et al.* (Fink et al., 2008) used a uniqueness analysis based on must-alias and liveness information to facilitate the verification of whether a summarized node represents more than one concrete reference.

The recency abstraction (Balakrishnan and Reps, 2006) is a simple and elegant technique that enables strong update by distinguishing the objects recently allocated from those created earlier. This approach allows strong update to be applied whenever a write access immediately follows an allocation, which is usually the case for initialization. Although the objective of recency abstraction is similar to targeted update, it uses a different abstraction that is not comparable to ours.

The issue of strong/weak update has been mostly studied for array structures. Cousot *et al.* (Cousot et al., 2010) proposed an efficient solution based on the ordering of array indexes. It may be not easy to generalize their method to the analysis of the pointer access. Fluid update (Dillig et al., 2010) is much closer to our approach. It is an abstract semantics that provides a sharp analysis for the array structure. The authors used bracket constraints to refine points-to information on arrays, which was shown to be effective to disambiguate array indexes. This approach was also extended in (Dillig et al., 2011) to deal with containers and other non-array structures.

10 Conclusion

Targeted update introduces a novel dimension in program analysis for tuning precision and efficiency. We have derived the abstract semantics from the concept of targets. This approach is validated on the benchmark suite SPECjvm98. The experimental results show a tangible precision enhancement compared with classical approaches while preserving a high scalability.

Acknowledgments. The author wishes to thank Laurent Mauborgne for his thoughtful feedback.

Bibliography

G. Balakrishnan and T.W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, pages 221–239, 2006.

B. Blanchet, P. Cousot, and R. Cousot. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003a.

B. Blanchet, P. Cousot, R. Cousot, et al. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003b.

D.R. Chase, M.N. Wegman, and F.K. Zadeck. Analysis of pointers and structures (with retrospective). In *Best of PLDI*, pages 343–359, 1990.

P.S. Chen, M.Y. Hung, Y.S. Hwang, et al. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *PPOPP*, pages 25–36, 2003.

A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

P. Cousot and R. Cousot. Comparing the Galois connection and widening/-narrowing approaches to Abstract interpretation. In *PLILP '92*, volume 631, pages 269–295, 1992.

P. Cousot, R. Cousot, and L. Mauborgne. A scalable segmented decision tree abstract domain. In *Pnueli Festschrift*, Lecture Notes in Computer Science, pages 72–95. Springer, 2010.

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.

I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.

I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200, 2011.

M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.

S.J. Fink, E. Yahav, N. Dor, et al. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

Z. Fu. *Static Analysis of Numerical Properties in the Presence of Pointers.* PhD thesis, Université de Rennes 1 – INRIA, Rennes, France, 2013.

Z. Fu. Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for java. In *VMCAI*, 2014.

D. Gopan, F. DiMaio, N. Dor, et al. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.

W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, pages 235–248, 1992.

T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.

M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.

R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.

R.P. Wilson and M.S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *PLDI*, pages 1–12, 1995.

A Elements of Abstract Interpretation

Abstract interpretation (Cousot and Cousot, 1977) is a theory of semantic approximation. The semantics of a program P can often be expressed by the least fixpoint $\mathbf{lfp}\,\mathsf{t}\,[\![P]\!]$ of a *transfer function* $\mathsf{t}\,[\![P]\!]$ over a partial ordered set $(A, \sqsubseteq)$, called the *semantic domain*. There may be different choices for the transfer function and semantic domain, depending on the level of precision we want to describe the program. In general, we have a very precise semantics $(A^\natural, \sqsubseteq^\natural)$, that describes exactly what a program does, called the *concrete* semantics, and a less precise but computable semantics $(A^\sharp, \sqsubseteq^\sharp)$, called the *abstract* semantics. The soundness of the abstract semantics is described using a *concretization function* $\gamma : A^\sharp \to A^\natural$, giving the meaning of the abstract elements in terms of concrete elements. We say that the abstract semantics $\mathbf{lfp}\,\mathsf{t}^\sharp\,[\![P]\!]$ is sound with respect to the concrete semantics $\mathbf{lfp}\,\mathsf{t}^\natural\,[\![P]\!]$, or say that the latter is approximated by the former, if $\mathbf{lfp}\,\mathsf{t}^\natural\,[\![P]\!] \sqsubseteq^\natural \gamma(\mathbf{lfp}\,\mathsf{t}^\sharp\,[\![P]\!])$.

In this paper, we frequently verify a stronger *soundness condition* in the form of

$$\mathsf{t}^\natural\,[\![P]\!] \circ \gamma \sqsubseteq^\natural \gamma \circ \mathsf{t}^\sharp\,[\![P]\!] \tag{31}$$

By "being sound", we always refer to the *partial soundness*, *i.e.*, if P terminates, then (31) holds.

B Definitions for $\mathsf{p}(\cdot)$ and $\mathsf{p}^\sharp(\cdot)$

Here, we give the precise definitions of the operators that resolve access paths. In the following, we use p and $\mathsf{p}^\sharp$ for elements of *Pter* and $Pter^\sharp$ respectively.

Definition 5. *Let* $\mathsf{p} = (\mathsf{p}_{env}, \mathsf{p}_{hp})$*. We say an access path* $\boldsymbol{u} \in Path_p$ resolves *to a reference* r*, or the reference can be reached by the access path following* p*, if one of the following rules is satisfied.*

$$\frac{\boldsymbol{u} = x \qquad \mathsf{p}_{env}(x) = r}{\mathsf{p}(\boldsymbol{u}) = r} \qquad \frac{\boldsymbol{u} = \boldsymbol{u}'.f \qquad \mathsf{p}(\boldsymbol{u}') = r \qquad \mathsf{p}_{hp}(r', f) = r}{\mathsf{p}(\boldsymbol{u}) = r} \tag{32}$$

Similar to the notation $\mathsf{p}(\boldsymbol{u})$*, we write* $\mathsf{p}^\sharp(\boldsymbol{u})$ *for the resolved abstract references of* $\boldsymbol{u} \in Path_p$*, called the* points-to set *of* $\boldsymbol{u}$ *under* $\mathsf{p}^\sharp$*. Let* $\mathsf{p}^\sharp = (\mathsf{p}^\sharp_{env}, \mathsf{p}^\sharp_{hp})$*,* $\mathsf{p}^\sharp(\boldsymbol{u})$ *is defined to be the smallest set satisfying:*

$$\frac{\boldsymbol{u} = x \qquad h \in \mathsf{p}^\sharp_{env}(x)}{h \in \mathsf{p}^\sharp(\boldsymbol{u})} \qquad \frac{\boldsymbol{u} = \boldsymbol{u}'.f \qquad h' \in \mathsf{p}^\sharp(\boldsymbol{u}') \qquad h \in \mathsf{p}^\sharp_{hp}(h', f)}{h \in \mathsf{p}^\sharp(\boldsymbol{u})} \tag{33}$$

For a numeric access path in the form of $\boldsymbol{u}_p.f_n$*, with* $\boldsymbol{u}_p \in Path_p$ *and* $f_n \in Fld_n$*, the definitions above can be extended as:*

$$\mathsf{p}(\boldsymbol{u}_p.f_n) = \mathsf{p}(\boldsymbol{u}_p), f_n \tag{34}$$

$$\mathsf{p}^\sharp(\boldsymbol{u}_p.f_n) = \{\delta_{h,f_n} \mid h \in \mathsf{p}^\sharp(\boldsymbol{u})\} \tag{35}$$

C Definition of γ_p

Definition 6. *The semantics of points-to graph is defined through the concretization function* $\gamma_p \in Pter^\sharp \to \wp(Pter)$*:*

$$\gamma_p(\mathsf{p}^\sharp) \triangleq \{\mathsf{p} \in Pter \mid \forall \boldsymbol{u} \in Path_p, \forall r \in Ref : \mathsf{p}(\boldsymbol{u}) = r \Rightarrow \triangleright(r) \in \mathsf{p}^\sharp(\boldsymbol{u})\} \tag{36}$$

For example, if a variable x resolves to r under $\mathsf{p} \in \gamma_p(\mathsf{p}^\sharp)$ and h is an abstraction of r, then h must be in the points-to set of x.

D Concrete semantics of $\texttt{Imp}_{np}$

As a shortcut, we set

$$D = Ref \times Fld_n \tag{37}$$

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

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

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

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

E Analysis Results for the Example in Sect. 7

Here, we show the analysis results of the example program in Fig. 2. The results are shown in Tab. 3. Column left shows the inferred constraints before each control point. Column right shows the Jimple statements. The part starting with "//" is our comments. In the constraints, δ_i for $i = 1, 2$ corresponds to $\delta_{h_i,val}$ of the example in Sect. 7. In the Jimple statements, r_2 (resp. r_3) corresponds to variables *hd* (resp. *node*) in the original program.

Table 3: Analysis results for the program in Fig. 2.

In	JimpleStmt
{true}	r0 := @parameter0: java.lang.String[]
{true}	$r1 = new List // allocate site h_1
{true}	specialinvoke $r1.< List: void < init>()>()
{true}	r2 = $r1
{true}	r2.< List: int val> = 0
$\{\delta_1 = 0\}$	r2.< List: List next> = null
$\{\delta_1 = 0\}$	i = −17
$\{\delta_1 = 0,\ i = -17\}$	goto [?= (branch)] // goto the loop entry
$\{i - \delta_1 = -17,\ -i \geq -42,\ i \geq -17\}$	$r4 = new List // allocate site h_2
$\{i - \delta_1 = -17,\ i \geq -17,\ -i \geq -42\}$	specialinvoke $r4.< List: void < init>()>()
$\{i - \delta_1 = -17,\ i \geq -17,\ -i \geq -42\}$	r3 = $r4
$\{i - \delta_1 = -17,\ i \geq -17,\ -i \geq -42\}$	r3.< List: int val> = i
$\{\delta_1 - \delta_2 = 17,\ i - \delta_1 = -17,\ \delta_1 \geq 0,\ -\delta_1 \geq -59\}$	$r5 = r2.< List: List next>
$\{\delta_1 - \delta_2 = 17,\ i - \delta_1 = -17,\ \delta_1 \geq 0,\ -\delta_1 \geq -59\}$	r3.< List: List next> = $r5
$\{\delta_1 - \delta_2 = 17,\ i - \delta_1 = -17,\ \delta_1 \geq 0,\ -\delta_1 \geq -59\}$	r2.< List: List next> = r3
$\{\delta_1 - \delta_2 = 17,\ i - \delta_1 = -17,\ \delta_1 \geq 0,\ -\delta_1 \geq -59\}$	tmp1 = r2.< List: int val>
$\{\delta_1 - \delta_2 = 17,\ i - \delta_2 = 0,\ \mathrm{tmp1} - \delta_2 = 17,\ \delta_2 \geq -17,\ -\delta_2 \geq -42\}$	tmp2 = tmp1 + 1
$\{\delta_1 - \delta_2 = 17,\ i - \delta_2 = 0,\ \mathrm{tmp1} - \delta_2 = 17,\ \mathrm{tmp2} - \delta_2 = 18,\ \delta_2 \geq -17,\ -\delta_2 \geq -42\}$	r2.< List: int val> = tmp2
$\{\delta_1 - \delta_2 = 18,\ i - \delta_2 = 0,\ \mathrm{tmp1} - \delta_2 = 17,\ \mathrm{tmp2} - \delta_2 = 18,\ \delta_2 \geq -17,\ -\delta_2 \geq -42\}$	i = i + 1
$\{i - \delta_1 = -17,\ -\delta_1 \geq -60,\ \delta_1 \geq 0\}$	if i ≤ 42 goto $r4 = new List //original loop entry
$\{i - \delta_1 = -17,\ -i \geq -43,\ i > 42\}$	return

F Proof Sketch of Thm. 1

Here, we give the proof sketch of Thm. 1.

Proof. Take an arbitrary $(\mathsf{n}, \mathsf{p}) \in \gamma_{\langle T\rangle}(\mathsf{n}^{\sharp}, \mathsf{p}^{\sharp})$, $\delta \in \mathsf{p}^{\sharp}(y_p.f_n)$, $ins \in Ins_{\mathsf{p}}\langle T\rangle$. Let $d = \mathsf{p}(y_p.f_n)$. We prove a stronger condition:

$$\xrightarrow{Num} (d = x_n)(\mathsf{n}) \models [ins](\llbracket \delta = x_n \rrbracket^{\eta(\delta)}(\mathsf{n}^{\sharp})) \tag{38}$$

Below we write $\mathsf{n}[d := \mathsf{n}(x_n)]$ to represent a mapping that is as n except that at d it takes the value of $\mathsf{n}(x_n)$; we write ins^{-1} for the inverse of ins. (Without loss of generality, we assume that ins is bijective, since it has to be injective) It suffices to prove

$$[ins^{-1}](\mathsf{n}[d := \mathsf{n}(x_n)]) \models [\![\delta = x_n]\!]^{\eta(\delta)}(\mathsf{n}^\sharp) \tag{39}$$

We make two cases following whether ins maps δ to d. Subsequently, the left hand side of the proof goal (39) is denoted by lhs.

- *Case I:* If ins maps δ to d, lhs can be written as $([ins^{-1}]\mathsf{n})[\delta := \mathsf{n}(x_n)]$. Since $[ins^{-1}]\mathsf{n} \models \mathsf{n}^\sharp$ by assumption, we obtain $lhs \models [\![\delta = x_n]\!]^\sharp_n$ following the soundness of $[\![\cdot]\!]^\sharp$ (Sect. 2).
- *Case II:* If ins does not map δ to d, we can then prove $lhs \models \mathsf{n}^\sharp$. This is because variable d does not appear in the free variables of $\mathsf{n}^\sharp$. (To be more clear, we always have, for any $\mathsf{n}, \mathsf{n}^\sharp$: $\mathsf{n} \models \mathsf{n}^\sharp \Rightarrow \mathsf{n}' \models \mathsf{n}^\sharp$ with n' being n restricted to the free variables of $\mathsf{n}^\sharp$. For example, $\{x \to 2, y \to 3\} \models \{x \geq 0\}$ $\Rightarrow \{x \to 2\} \models \{x \geq 0\}$.)

Following Lem. 1, if (TU) holds, the second case can be ruled out. If (TU) is not satisfied, we have to join the two cases, obtaining $lhs \models [\![\delta = x_n]\!]^\sharp_n(\mathsf{n}^\sharp) \sqcup \mathsf{n}^\sharp$. $\square$