Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 287 (2012) 53–64

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Generic Abstraction of Dictionaries and Arrays

Jędrzej Fulara[1,2]

Institute of Informatics
University of Warsaw

Abstract

We present a generic abstract domain for analysis of dictionary and array content. Our technique is parametrised by the abstractions of scalars, dictionary keys and dictionary values. It can be instantiated with various existing domains, including non-numerical ones (such as domains for analysis of properties of string variables). It is powerful enough to express relations between container content and scalars.
The analysis is fully automatic. The container is partitioned according to properties of the keys, captured by the underlying key abstraction. The precision and cost of the analysis are customisable and depend on the choice of the abstractions of keys, dictionary elements and scalar variables.
We show examples in which the technique is used to reason about arrays as well as string-keyed dictionaries. The approach was also experimentally evaluated.

Keywords: Abstract Interpretation, Abstract Domains, Dictionary Content Analysis

Collections such as dictionaries and arrays are very important building blocks of programs, thus static analysis techniques should be able to reason about the content of such containers. Our goal is to provide a generic solution for modelling arbitrary dictionaries and arrays in static analysis by abstract interpretation [2]. The technique should be fully automatic and it should be possible to adjust its precision/cost ratio. It should be possible to instantiate the technique not only with numerical abstract domains [9,13,15], but also with domains of other types, such as e.g. domains for string analysis [8].

Abstract Interpretation We use the classic definition of abstract interpretation [2]. An abstract domain is a tuple $A = \langle \mathcal{A}, \sqcap_{\mathsf{a}}, \sqcup_{\mathsf{a}}, \alpha_{\mathsf{a}}, \gamma_{\mathsf{a}}, \delta_{\mathsf{a}}, \nabla_{\mathsf{a}} \rangle$ denoting the set of abstract states, meet, join, abstraction, concretisation, transfer function and widening. We require also a projection $\cdot\downarrow_v$ (called *variable elimination*), a dual operator $\cdot\uparrow_v$ (*variable introduction*) and a forget operator $\cdot\updownarrow_v$.

[1] This work has been partially supported by the Polish Ministry of Science and Higher Education, grant N206 493138

[2] Email: fulara@mimuw.edu.pl

doi:10.1016/j.entcs.2012.09.006

Notation We analyse programs that operate over a set of scalar variables $\mathcal{V}ar$ (with values in some set $\mathbb{V}$) and dictionaries $\mathcal{V}ar_c$. The dictionary keys and elements belong to sets $\mathbb{K}$ and $\mathbb{E}$, respectively. A concrete program state is a pair $(\rho, \tau) \in (\mathcal{V}ar \to \mathbb{V}) \times (\mathcal{V}ar_c \to (\mathbb{K} \rightharpoonup \mathbb{E}))$. The only dictionary statements that we consider are updates $T[v_1] \leftarrow v_2$ and $T[v_1] \leftarrow c$, read $v_2 \leftarrow T[v_1]$ and boolean predicates $\phi(T[v_1], v_2)$, where $T \in \mathcal{V}ar_c$, $v_1, v_2 \in \mathcal{V}ar$ and $c \in \mathbb{E}$.

Outline Section 1 introduces a structure to represent a single dictionary. In Section 2 we show how to use it to build an abstract domain. Examples of the domain are shown in Section 3. Section 4 describes an experiment. Related work is sketched in Section 5. We conclude in Section 6. Proofs and additional examples can be found in an extended version of this work [8].

1 Representation of a Dictionary

We model a dictionary $T \in \mathcal{V}ar_c$ as a set of *abstract segments*. Each abstract segment represents some set of (concrete) keys and corresponding dictionary elements. Let K and V be two abstract domains (with carriers $\langle \mathcal{K}, \sqcup_k, \sqcap_k \rangle$ and $\langle \mathcal{V}, \sqcup_v, \sqcap_v \rangle$). An abstract segment is a pair $(k, v) \in \mathcal{K} \times \mathcal{V}$, where k models a set of concrete keys (together with their relations to scalars) and is called an *abstract key*, while v abstracts the dictionary elements (and their relations to scalars) and is called an *abstract value*. The dictionary abstraction is a (restricted) powerset of the product $K \times V$ [3].

Let us introduce some auxiliary terminology. In a lattice $\langle \mathcal{A}, \sqcup_a, \sqcap_a, \bot_a, \top_a \rangle$, $a \in \mathcal{A}$ is *empty*, if $a = \bot_a$; $a \in \mathcal{A}$ *overlaps* with $b \in \mathcal{A}$, when $a \sqcap_a b \neq \bot_a$.

We define now a lattice $\langle \mathcal{D}, \sqcup_d, \sqcap_d \rangle$, where $\mathcal{D} \subseteq \mathcal{P}_{fin}(\mathcal{K} \times \mathcal{V})$ and each $d \in \mathcal{D}$ fulfils the following additional *well-formedness conditions*:

(i) for $(k_1, v_1) \in d$ and $(k_2, v_2) \in d$ either $(k_1, v_1) = (k_2, v_2)$ or $k_1 \sqcap_k k_2 = \bot_k$,

(ii) for $(k, v) \in d$, $k \neq \bot_k$ and $v \neq \bot_v$.

The first condition states that every two abstract segments represent disjoint sets of concrete elements. The second condition forbids abstract segments with empty abstract keys or abstract values. An abstract segment $(\bot_k, v)$ would represent an empty fragment of the dictionary, while $(k, \bot_v)$ would model a set of elements that could not have been initialised to any value. Such abstract segments are superfluous in our representation.

For each (concrete) dictionary $d \colon \mathbb{K} \rightharpoonup \mathbb{E}$ represented by an abstract dictionary $d \in \mathcal{D}$ and for each key $n \in Dom(d)$, there exists an abstract segment $(k, v) \in d$ such that n and $d(n)$ are abstracted by k and v, respectively.

Meet and Join The meet $a \sqcap_d b$ of $a, b \in \mathcal{D}$ consists of abstract segments obtained as a point-wise meet of some overlapping segments from a and b:

$$\{(k_a \sqcap_k k_b, v_a \sqcap_v v_b) \mid (k_a, v_a) \in a, (k_b, v_b) \in b, k_a \sqcap_k k_b \neq \bot_k, v_a \sqcap_v v_b \neq \bot_v\} .$$

Lemma 1.1 *The meet operator is well defined, i.e.* $\mathcal{a} \sqcap_{\mathsf{d}} \mathcal{b} \in \mathcal{D}$.

The join $\mathcal{a} \sqcup_{\mathsf{d}} \mathcal{b}$ of $\mathcal{a}, \mathcal{b} \in \mathcal{D}$ should represent all concrete dictionaries abstracted by $\mathcal{a}$ or $\mathcal{b}$, thus one could try to define it as a union $\mathcal{a} \cup \mathcal{b}$. However, join defined in this way could violate the first well-formedness condition. We show now how to avoid this problem, i.e. transform an arbitrary set of abstract keys into a set in which no two keys overlap. The idea is to identify groups of overlapping keys and replace each group with its least upper bound.

Let $\langle \mathcal{A}, \sqcap_{\mathsf{a}}, \sqcup_{\mathsf{a}} \rangle$ be a complete lattice and let S be a finite subset of $\mathcal{A}$.

Definition 1.2 We say that a finite family of non-empty sets $\mathcal{X} = \{X_1, X_2, \ldots, X_k\}$, where each $X_i \subseteq S$, is a *disjoint partition* of S, iff:

- $\mathcal{X}$ is a partition of S (i.e. $S = \bigcup \mathcal{X}$ and $X_i \cap X_j = \emptyset$ for $i \neq j$),
- for every $X_i, X_j \in \mathcal{X}$, where $i \neq j$, it holds that $(\bigsqcup_{\mathsf{a}} X_i) \sqcap_{\mathsf{a}} (\bigsqcup_{\mathsf{a}} X_j) = \bot_{\mathsf{a}}$.

A disjoint partition of S always exists — if $S = \emptyset$, then $\mathcal{X} \triangleq \emptyset$, otherwise one can take $\mathcal{X} \triangleq \{S\}$. If S has multiple disjoint partitions, then we are interested in a partition $\mathcal{X}$ of S that does not perform any unnecessary grouping:

Definition 1.3 We say that a disjoint partition $\mathcal{C}$ of S is *least*, if for any disjoint partition $\mathcal{X}$ of S it holds that $\forall_{C \in \mathcal{C}} \exists_{X \in \mathcal{X}} C \subseteq X$.

Intuitively, the least disjoint partition groups (puts into the same X_i) these elements of S that must be grouped together in each disjoint partition of S.

Lemma 1.4 *The least disjoint partition of S exists and is uniquely defined.*

We use the concept of the least disjoint partition to transform arbitrary $\mathcal{c} \in \mathcal{P}_{\mathit{fin}}(\mathcal{K} \times \mathcal{V})$ into an abstract dictionary $\mathcal{d} \in \mathcal{D}$. Let S denote the set of abstract keys of "non-empty" abstract segments $(k, v) \in \mathcal{c}$:

$$S \triangleq \{k \mid (k, v) \in \mathcal{c}, k \neq \bot_{\mathsf{k}}, v \neq \bot_{\mathsf{v}}\} .$$

Let $\mathcal{C}$ denote the least disjoint partition of S. We define a *disjoint normalisation* function $dNorm \colon \mathcal{P}_{\mathit{fin}}(\mathcal{K} \times \mathcal{V}) \to \mathcal{D}$ as:

$$dNorm(\mathcal{c}) \triangleq \{(\bigsqcup_{\mathsf{k}}{}_{j} k_j, \bigsqcup_{\mathsf{v}}{}_{j} v_j) \mid \{k_1, \ldots, k_m\} \in \mathcal{C}, (k_j, v_j) \in \mathcal{c}, j \in \{1, \ldots, m\}\} .$$

This normalisation can be computed in $O(|\mathcal{c}|^3)$ time, including the computation of the least disjoint partition $\mathcal{C}$. Now the join $\mathcal{a} \sqcup_{\mathsf{d}} \mathcal{b}$ can be defined just as the normalised union $\mathcal{a} \cup \mathcal{b}$, i.e. $\mathcal{a} \sqcup_{\mathsf{d}} \mathcal{b} \triangleq dNorm(\mathcal{a} \cup \mathcal{b})$.

Theorem 1.5 *Set* $\mathcal{D}$ *forms a lattice under* $\sqcap_{\mathsf{d}}$ *and* $\sqcup_{\mathsf{d}}$.

Widening The widening preserves all segments from the first argument that are disjoint with each segment from the second argument, replaces overlapping segments with their widening in K and V and replaces each segment (l, w) from the second argument that is disjoint with all segments from the first one with $(\top_{\mathsf{k}}, w)$ [8].

2 The Domain

We utilise now the lattice defined above to define an abstract domain. The content of each dictionary will be over-approximated using $\langle \mathcal{D}, \sqcup_{\mathsf{d}}, \sqcap_{\mathsf{d}} \rangle$ defined above. In an abstract segment (k, v), k and v over-approximate sets of keys of concrete elements and of their possible values, respectively. If a key is not abstracted at all, then the corresponding element cannot be initialised.

The abstract keys are modelled within an abstract domain K over the set $\mathcal{Var} \cup \{v_k\}$, where v_k is an artificial *key variable* used to represent the value of a key. Similarly, the abstract values are represented using an abstract domain V over the set $\mathcal{Var} \cup \{v_v\}$, where v_v is a *value-tracking variable* that represents the value of a dictionary element. In this approach it is possible to express relations between scalars and keys as well as scalars and dictionary elements.

The lattice $\mathcal{D}(\mathcal{K}, \mathcal{V})$ can be used to over-approximate the content of a dictionary, but it does not give any information about which elements must be initialised. We solve this problem by associating with each dictionary also an element of $\mathcal{D}(\mathcal{K}, \mathcal{Bool})$ that is used to over-approximate the set of uninitialised dictionary elements. A segment (k, True) means that elements at keys abstracted by k may be uninitialised. If some key is not abstracted by any segment, then the element at this key must be initialised. As False is the bottom in the lattice of booleans, segments (l, False) are automatically removed.

The scalar part of the state is abstracted in some abstract domain A over the set of variables $\mathcal{Var}$. We require also conversion functions $\kappa_{A \to K}$ and $\kappa_{K \to A}$ between the domains A and K as well as $\kappa_{A \to V}$ and $\kappa_{V \to A}$ between A and V.

Now we may define the domain $C = \langle \mathcal{C}, \sqcup_{\mathsf{c}}, \sqcap_{\mathsf{c}}, \gamma_{\mathsf{c}}, \alpha_{\mathsf{c}}, \delta_{\mathsf{c}}, \nabla_{\mathsf{c}} \rangle$. The set of abstract states $\mathcal{C}$ is given by $\mathcal{C} \triangleq \mathcal{A} \times (\mathcal{Var}_c \to \mathcal{D}(\mathcal{K}, \mathcal{V})) \times (\mathcal{Var}_c \to \mathcal{D}(\mathcal{K}, \mathcal{Bool}))$. An abstract state will be denoted as $(a, d, i) \in \mathcal{C}$.

Domain Operations The meet $\sqcap_{\mathsf{c}}$ and join $\sqcup_{\mathsf{c}}$ are defined point-wise. The widening ∇_{c} is defined in a lazy manner, as the dictionary modifications should get stable, when the abstract state $a \in \mathcal{A}$ of scalars stabilises:

$$(a_1, d_1, i_1) \nabla_{\mathsf{c}} (a_2, d_2, i_2) \triangleq \begin{cases} (a_1 \nabla_{\mathsf{a}} a_2, d_1 \sqcup_{\mathsf{d}} d_2, i_1 \sqcup_{\mathsf{d}} i_2) & \text{if } a_1 \nabla_{\mathsf{a}} a_2 \neq a_1 \\ (a_1 \nabla_{\mathsf{a}} a_2, d_1 \nabla_{\mathsf{d}} d_2, i_1 \nabla_{\mathsf{d}} i_2) & \text{otherwise.} \end{cases}$$

Concretisation The concretisation of scalars is defined using the concretisation γ_{a} in the scalar domain A. Let us consider a concrete valuation $\rho \colon \mathcal{Var} \to \mathbb{V}$ of scalar variables, a dictionary $T \in \mathcal{Var}_c$. and a concrete key $n \in \mathbb{K}$. If there is an abstract segment $(k, \mathsf{True}) \in i(T)$ such that n is abstracted by k, then $T[n]$ may be uninitialised. If there is an abstract segment $(l, w) \in d(T)$ such that n is abstracted by l, then $T[n]$ may have some value abstracted by w. If there is neither $(k, \mathsf{True}) \in i(T)$ nor $(l, w) \in d(T)$ such that n is abstracted by k or l, then $T[n]$ can be neither initialised nor uninitialised, hence for the valuation ρ of scalars, there is no valuation $\tau \colon \mathcal{Var}_c \to (\mathbb{K} \rightharpoonup \mathbb{E})$ of dictionaries.

Following these observations, we define a predicate $\mathfrak{I}(\rho, T, n, i)$ for $\rho \colon \mathcal{Var} \to \mathbb{V}$,

$$\frac{\begin{array}{c} d' \triangleq \lambda T.dNorm\Big(\{(\delta_{\mathsf{k}}(I,k),\delta_{\mathsf{v}}(I,v)) \mid (k,v) \in d(T)\}\Big) \\ i' \triangleq \lambda T.dNorm\Big(\{(\delta_{\mathsf{k}}(I,k),\mathsf{True}) \mid (k,\mathsf{True}) \in i(T)\}\Big) \end{array}}{\delta_{\mathsf{c}}\Big(I,(a,d,i)\Big) \triangleq \Big(\delta_{\mathsf{a}}(I,a),d',i'\Big)}$$

Fig. 1. Transfer function for a scalar instruction I

$T \in \mathcal{V}ar_c$, $n \in \mathbb{K}$ and the initialisation part of the abstract state i that holds if and only if $T[n]$ may be uninitialised:

$$\mathfrak{I}(\rho,T,n,i) \triangleq \mathsf{True} \Leftrightarrow \exists_{(k,\mathsf{True})\in i(T)}\, \exists_{\sigma\in\gamma_{\mathsf{k}}(k)}\, \sigma_{|\mathcal{V}ar} = \rho, \sigma(v_k) = n\ . \tag{1}$$

Similarly, we define a predicate $\mathfrak{V}(\rho,T,n,m,d)$ for $\rho\colon \mathcal{V}ar \to \mathbb{V}$, $T \in \mathcal{V}ar_c$, $n \in \mathbb{K}$, $m \in \mathbb{E}$ that holds when $T[n]$ may be equal to m:

$$\begin{aligned} \mathfrak{V}(\rho,T,n,m,d) \triangleq \mathsf{True} \Leftrightarrow \exists_{(k,v)\in d(T)}\, \exists\sigma_k \in \gamma_{\mathsf{k}}(k)\, \exists\sigma_v \in \gamma_{\mathsf{v}}(v) \\ \sigma_{k|\mathcal{V}ar} = \sigma_{v|\mathcal{V}ar} = \rho, \sigma_k(v_k) = n, \sigma_v(v_v) = m\ . \end{aligned} \tag{2}$$

This allows us to define the concretisation $\gamma_{\mathsf{c}}((a,d,i))$ as:

$$\Big\{(\rho,\tau) \mid \rho \in \gamma_{\mathsf{a}}(a), \forall_{T\in\mathcal{V}ar_c} \forall_{n\in\mathbb{K}}\ (n \notin Dom(\tau(T)) \text{ and } \mathfrak{I}(\rho,T,n,i)) \text{ or } \\ (n \in Dom(\tau(T)) \text{ and } \mathfrak{V}(\rho,T,n,\tau(T)(n),d))\Big\}\ .$$

2.1 Transfer Function

We provide the transfer functions for scalar and dictionary statements. We illustrate the definitions on examples, where all A, K and V are chosen as the domain of intervals with values in $\mathbb{Z}$. For clarity, in each abstract segment we show only the values of the key and value-tracking variables v_k and v_v.

Scalar Statements The abstract keys and abstract values model relationships with scalar variables, thus scalar statements are interpreted not only in the scalar domain A, but also in all abstract segments in all containers, as shown in Figure 1. When a scalar variable is modified, all its relations to abstract keys and abstract dictionary elements are updated.

Dictionary Statements An empty dictionary is created by $T \leftarrow$ **new dict**:

$$\delta_{\mathsf{c}}(T \leftarrow \textbf{new dict}, (a,d,i)) \triangleq (a, d[T \mapsto \emptyset], i[T \mapsto \{(\top_{\mathsf{k}},\mathsf{True})\}])\ .$$

We proceed now with a dictionary read $v_2 \leftarrow T[v_1]$, where $T \in \mathcal{V}ar_c$ and $v_1, v_2 \in \mathcal{V}ar$ (see Figure 2). Intuitively, we retrieve from $d(T)$ all abstract segments (there may be more than one) whose keys overlap with the key for the access $T[v_1]$. Formally, we compute $k \in \mathcal{K}$ by adding to a the artificial key variable v_k, assigning $v_k \leftarrow v_1$ and converting the result into the domain K, i.e. $k \triangleq \kappa_{A\to K}(\delta_{\mathsf{a}}(v_k \leftarrow v_1, a{\uparrow_{v_k}}))$. We take the join of all values in all abstract segments $(l,w) \in d(T)$ whose keys overlap with k, i.e. $v \triangleq \bigsqcup_{\mathsf{v}}\{w \mid (l,w) \in d(T), l \sqcap_{\mathsf{k}} k \neq \bot_{\mathsf{k}}\}$, convert v back to the domain A, assign $v_2 \leftarrow v_v$ and eliminate the special value-tracking variable v_v. Finally, we invalidate

$$\frac{\begin{array}{c} k \triangleq \kappa_{A\to K}\Big(\delta_{\mathsf{a}}(v_k \leftarrow v_1, a\!\upharpoonleft_{v_k})\Big) \qquad v \triangleq \bigsqcup_{\mathsf{v}}\Big\{w \mid (l,w) \in d(T), k \sqcap_{\mathsf{k}} l \neq \bot_{\mathsf{k}}\Big\} \\ a' \triangleq \delta_{\mathsf{a}}(v_2 \leftarrow v_v, \kappa_{V\to A}(v) \sqcap_{\mathsf{a}} a\!\upharpoonleft_{v_v}) \qquad (\_, d', i') \triangleq (a,d,i)\!\updownarrow_{v_2} \end{array}}{\delta_{\mathsf{c}}\Big(v_2 \leftarrow T[v_1], (a,d,i)\Big) \triangleq (a'\!\downharpoonleft_{v_v}, d', i')}$$

Fig. 2. Transfer rule for $v_2 \leftarrow T[v_1]$

$$\frac{\begin{array}{c} k \triangleq \kappa_{A\to K}\Big(\delta_{\mathsf{a}}(v_k \leftarrow v_1, a\!\upharpoonleft_{v_k})\Big) \qquad v \triangleq \kappa_{A\to V}\Big(\delta_{\mathsf{a}}(v_v \leftarrow v_2, a\!\upharpoonleft_{v_v})\Big) \\ x \triangleq dNorm\Big(d(T) \cup \{(k,v)\}\Big) \end{array}}{\delta_{\mathsf{c}}\Big(T[v_1] \leftarrow v_2, (a,d,i)\Big) \triangleq (a, d[T \mapsto x], i)}\ \neg\mathfrak{S}_{\mathsf{k}}(k)$$

Fig. 3. Weak update $T[v_1] \leftarrow v_2$

the old value of v_2 in all abstract segments. The read results in an error, whenever the accessed key may be uninitialised, i.e. $\{l \mid (l, \mathsf{True}) \in i(T), k \sqcap_{\mathsf{k}} l \neq \bot_{\mathsf{k}}\} \neq \emptyset$.

Example 2.1 Let us consider $v_1 \in \mathcal{V}ar$ with $a(v_1) = [1,4]$ and $T \in \mathcal{V}ar_c$ modelled as $d(T) = \{([0,2],[-2,1]),([3,5],[4,4]),([6,9],[2,7])\}$ and $i(T) = \{([8,\infty], \mathsf{True})\}$. The read $v_2 \leftarrow T[v_1]$ gives $a(v_2) = [-2,4]$.

We define both weak and strong dictionary updates. The strong update can be performed only when for each valuation of the scalars there is at most one possible value of the updated key. We formalise this by defining the following unary predicate $\mathfrak{S}$ on the domain K:

$$\mathfrak{S}(k) = \mathsf{True} \Leftrightarrow \forall_{\sigma_1,\sigma_2 \in \gamma_{\mathsf{k}}(k)} (\sigma_{1|\mathcal{V}ar} = \sigma_{2|\mathcal{V}ar} \Rightarrow \sigma_1(v_k) = \sigma_2(v_k)) \ .$$

This definition is not very practical, thus we require that the domain K is equipped with a domain-specific predicate $\mathfrak{S}_{\mathsf{k}}$ that implies $\mathfrak{S}$, i.e. $\forall_{k \in \mathcal{K}} \mathfrak{S}_{\mathsf{k}}(k) \Rightarrow \mathfrak{S}(k)$. If $\mathfrak{S}_{\mathsf{k}}(k) = \mathsf{True}$, then we say that k is a *singleton.*

Let us consider an update $T[v_1] \leftarrow v_2$. We compute the abstract key $k \in \mathcal{K}$ in the same way as in the read. Similarly we obtain the abstract value $v \in \mathcal{V}$.

If k is not a singleton (i.e. $\neg\mathfrak{S}_{\mathsf{k}}(k)$) then a weak update is performed as defined in Figure 3. We add the new abstract segment (k,v) to $d(T)$ and compute $dNorm(d(T) \cup \{(k,v)\})$. The initialisation information is not altered.

If k is a singleton, a strong update can be performed. The container $d(T)$ may already contain an abstract segment (l,w) that describes the updated element, i.e. $k \sqcap_{\mathsf{k}} l \neq \bot_{\mathsf{k}}$. The new value should be assigned only to the modified element, all other elements associated with keys abstracted by l should remain unchanged. We need to split l into a collection of smaller keys $k, m_1, m_2, \ldots m_j$ which represent together the same concrete keys as l. We say that a function $\zeta\colon \mathcal{K} \times \mathcal{K} \to \mathcal{P}(\mathcal{K})$ is a *decomposition* of an abstract key $l \in \mathcal{K}$ with respect to a singleton $k \in \mathcal{K}$ if:

- $\forall_{m_1,m_2 \in \zeta(l,k)\cup\{k\}}\, m_1 \neq m_2 \Rightarrow m_1 \sqcap_{\mathsf{k}} m_2 = \bot_{\mathsf{k}}$,
- $k \notin \zeta(l,k)$,
- $\gamma_{\mathsf{k}}(k) \cup \big(\bigcup_{m \in \zeta(l,k)} \gamma_{\mathsf{k}}(m)\big) = \gamma_{\mathsf{k}}(l)$.

The definition of $\zeta(l,k)$ must be provided together with the domain K.

We define an operation $\uplus\colon \mathcal{D}(\mathcal{K},\mathcal{V}) \times (\mathcal{K} \times \mathcal{V}) \rightharpoonup \mathcal{D}(\mathcal{K},\mathcal{V})$ so that $\mathcal{d} \uplus (k,v)$

$$\frac{\begin{array}{c} k \triangleq \kappa_{A\to K}\Big(\delta_{\mathsf{a}}(v_k \leftarrow v_1, a{\upharpoonleft}_{v_k})\Big) \qquad v \triangleq \kappa_{A\to V}\Big(\delta_{\mathsf{a}}(v_v \leftarrow v_2, a{\upharpoonleft}_{v_v})\Big) \\ x \triangleq d(T) \uplus (k, v) \qquad y \triangleq i(T) \uplus (k, \mathsf{False}) \end{array}}{\delta_{\mathsf{c}}\Big(T[v_1] \leftarrow v_2, (a, d, i)\Big) \triangleq \Big(a, d[T \mapsto x], i[T \mapsto y]\Big)}\ \mathfrak{S}_{\mathsf{k}}(k)$$

Fig. 4. Strong update $T[v_1] \leftarrow v_2$

$$\frac{\begin{array}{c} k \triangleq \kappa_{A\to K}\Big(\delta_{\mathsf{a}}(v_k \leftarrow v_1, a{\upharpoonleft}_{v_k})\Big) \qquad v \triangleq \bigsqcup_{\mathsf{v}}\Big\{w \mid (l, w) \in d(T), k \sqcap_{\mathsf{k}} l \neq \bot_{\mathsf{k}}\Big\} \\ (v_t, v_f) \triangleq \pi_{\mathsf{v}}\Big(\phi(v_v, v_2), v\Big) \quad x_t \triangleq d(T) \uplus (k, v_t) \qquad x_f \triangleq d(T) \uplus (k, v_f) \end{array}}{\delta_{\mathsf{c}}\Big(\phi(T[v_1], v_2), (a, d, i)\Big) \triangleq (a, d[T \mapsto x_t], i), (a, d[T \mapsto x_f], i)}\ \mathfrak{S}_{\mathsf{k}}(k)$$

Fig. 5. Boolean predicate $\phi(T[v_1], v_2)$

overwrites in $\mathcal{d}$ the elements at keys abstracted by k:

$$\mathcal{d} \uplus (k, v) \triangleq \big(\mathcal{d} \setminus \{(l, w) \mid (l, w) \in \mathcal{d}, l \sqcap_{\mathsf{k}} k \neq \bot_{\mathsf{k}}\}\big) \cup \{(k, v)\}$$
$$\cup \{(m, w) \mid (l, w) \in \mathcal{d}, l \sqcap_{\mathsf{k}} k \neq \bot_{\mathsf{k}}, m \in \zeta(l, k)\}\ .$$

The operation $\mathcal{d} \uplus (k, v)$ is defined only if k is a singleton.

Let k and v be as in the weak update and let k be a singleton. The strong update overwrites in $d(T)$ the old value associated with the abstract key k and marks in $i(T)$ that the element at key abstracted by k must be initialised (see Figure 4). The strong update forgets only the old value of the updated element (by the definition of the decomposition) and replaces it with an abstract value that over-approximates the inserted concrete value.

Example 2.2 [Weak update] Consider the same container $T \in \mathcal{Var}_c$ and scalar $v_1 \in \mathcal{Var}$ as in Example 2.1, with an additional scalar $v_2 \in \mathcal{Var}$ such that $a(v_2) = [8, 8]$. The update $T[v_1] \leftarrow v_2$ gives $d(T) = \{([0, 5], [-2, 8]), ([6, 9], [2, 7])\}$.

Example 2.3 [Strong update] Consider scalars $v_1, v_2 \in \mathcal{Var}$, such that $a(v_1) = [2, 2]$ and $a(v_2) = [7, 7]$ and a container $T \in \mathcal{Var}_c$ with $d(T) = \{([0, 5], [1, 3])\}$, $i(T) = \{([0, \infty], \mathsf{True})\}$. The strong update $T[v_1] \leftarrow v_2$ modifies T so that $d(T) = \{([0, 1], [1, 3]), ([2, 2], [7, 7]), ([3, 5], [1, 3])\}$. It also marks that the updated element must be initialised, setting $i(T) = \{([0, 1], \mathsf{True}), ([3, \infty], \mathsf{True})\}$.

Boolean Predicates Boolean predicates (that occur in conditional statements) may operate over a scalar variable and a dictionary access, e.g. $\phi(T[v_1], v_2)$. We restrict the possible values of $T[v_1]$ as shown in Figure 5. Let k denote the abstract key for the access $T[v_1]$ and v is the corresponding abstract value. If k is a singleton, then we restrict v according to $\phi(v_v, v_2)$.

Example 2.4 Consider $v_1, v_2 \in \mathcal{Var}$ such that $a(v_1) = [4, 4]$ and $a(v_2) = [0, 0]$ and $T \in \mathcal{Var}_c$ such that $d(T) = \{([0, 4], [-2, 5])\}$. A test $T[v_1] \leq v_2$ evaluates to two abstract states $(\mathcal{c}_{\mathsf{True}}, \mathcal{c}_{\mathsf{False}})$, in which $d(T)$ is given by $\{([0, 3], [-2, 5]), ([4, 4], [-2, 0])\}$ and $\{([0, 3], [-2, 5]), ([4, 4], [1, 5])\}$.

3 Examples

We present examples, in which the domain is used to reason about arrays and string-keyed dictionaries.

3.1 Analysis of Arrays

Let us discuss a partial array initialisation presented in Figure 6(a). Our analysis detects that after this code fragment first j elements of T are initialised.

In this example, we use a very simple relational domain of upper bounds B [8], in which the set of abstract states $\mathcal{B}$ is a map $\mathcal{Var} \to \mathcal{P}(\mathcal{Var} \times \{<, \leq\})$. Intuitively, for each variable x, we keep the set of variables greater than x (with an additional indicator whether the inequality is strict).

We instantiate the domain C by fixing $A = B(\mathcal{Var})$, $K = B(\mathcal{Var} \cup \{v_k\})$ and $V = B(\mathcal{Var} \cup \{v_v\})$. Below we do not write all bounds explicitly. Instead, for each abstract key k we show only constraints for the key variable v_k. If $(x, \lhd) \in k(v_k)$, we write "$\lhd x$". If $(v_k, \lhd) \in k(y)$, then we write "$\rhd y$". If $(z, \leq) \in k(v_k)$ and $(v_k, \leq) \in k(z)$, then we use a shortcut "$=z$". For each abstract value v we show only constraints with the value-tracking variable v_v. We also assume that in $\mathcal{Var}$ there is a special variable v_0 that is equal to 0.

The statement $T \leftarrow$ **new array**(n) creates a new array, whose indices range over $[0, n)$. Note that at this program point $j = 0$ and $T.length = n$, thus the range of indices l of uninitialised array elements can be written as $v_0 = j \leq l < n = T.length$. This observation justifies the abstract state just before the loop, which is $d(T) = \emptyset$ and $i(T) = \{(\{<n, <T.length, \geq j, \geq v_0\}, \mathsf{True})\}$.

We assume that nothing can be statically determined about the test $\phi(x)$. Let us focus now on the array modification $T[j] \leftarrow x$ in line 5. The abstract key $k \in \mathcal{K}$ for the array access $T[j]$ contains the constraints $v_k \leq j$ and $j \leq v_k$, thus k is a singleton and the strong update is performed. The inserted abstract segment is $(k, v) = (\{=j, <n, \geq v_0\}, \{=x\})$ (for sake of clarity, we loose here the constraints not important in the analysis). After this update, the content of the array is modelled as $d(T) = \{(\{=j, <n, \geq v_0\}, \{=x\})\}$. The initialisation information is $i(T) = \{(\{=j, <n, \geq v_0\}, \mathsf{False}), (\{<n, >j\}, \mathsf{True})\}$.

```
1: $x \leftarrow 0$, $j \leftarrow 0$, $T \leftarrow$ **new array**$(n)$
2: **while** $x < n$ **do**
3:     $x \leftarrow x + 1$
4:     **if** $\phi(x)$ **then**
5:         $T[j] \leftarrow x$, $j \leftarrow j + 1$
6:     **end if**
7: **end while**
```

(a)

```
1:  at ← "b"
2:  **repeat**
3:      setattr(obj, at, 6)
4:      at ← at + "c"
5:  **until** random() = False
6:  **if** random() = True **then**
7:      obj.x ← 8
8:  **else**
9:      obj.x ← "text"
10: **end if**
11: **print** obj.b - 1
12: **print** obj.bcc - 1
13: **print** obj.x - 1
```

(b)

Fig. 6. Partial array initialisation (6(a)) and dynamically added attributes (6(b))

After the statement $j \leftarrow j+1$, $d(T)$ and $i(T)$ are equal to

$$\{(\{<j, <n, \geq v_0\}, \{=x\})\} \quad \text{and} \quad \{(\{<j, <n, \geq v_0\}, \mathsf{False}), (\{<n, \geq j\}, \mathsf{True})\} .$$

In the second loop iteration, $x \leftarrow x+1$ modifies $d(T)$ so that $d(T) = \{(\{<j, <n, \geq v_0\}, \{<x\})\}$ ($i(T)$ remains unchanged) and $T[j] \leftarrow x$ results in

$$\begin{aligned} d(T) =& \Big\{ (\{<j, <n, \geq v_0\}, \{<x\}), (\{=j, <n, \geq v_0\}, \{=x\}) \Big\} \\ i(T) =& \Big\{ (\{<j, <n, \geq v_0\}, \mathsf{False}), (\{=j, <n, \geq v_0\}, \mathsf{False}), (\{<n, >j\}, \mathsf{True}) \Big\} . \end{aligned}$$

Finally, after $j \leftarrow j+1$ we get $d(T)$ and $i(T)$ equal to:

$$\{(\{<j, <n, \geq v_0\}, \{\leq x\})\} \quad \text{and} \quad \{(\{<j, <n, \geq v_0\}, \mathsf{False}), (\{<n, \geq j\}, \mathsf{True})\} .$$

When analysing the next loop iteration, it turns out that this is already the loop invariant. The first abstract segment in $i(T)$ guarantees that all elements at indices smaller than j were initialised, while $d(T)$ ensures that all values of these elements are smaller than or equal to x.

3.2 Dictionaries

In this example we focus on string-keyed dictionaries. This example is inspired by the representation of objects in dynamic programming languages (such as Python): an object is stored as a dictionary, where each entry represents an attribute of the object. Attributes can be added and removed during program execution. Different types of values may be assigned to the same attribute during the program execution. For instance `obj.attr` may be at some point an integer, at another point a string, while somewhere else it may refer to a function. This flexibility, although sometimes useful and convenient, leads often to serious errors. When a missing attribute is accessed, the program fails with an `AttributeError`. When an attribute does not match the expected type (e.g. a string is encountered in an arithmetic operation), a runtime `TypeError` is raised. We show how to use our technique to statically detect such problems.

Attribute Analysis We abstract each attribute of an object by its possible type. For simplicity, we consider only integer and string attributes. The types can be abstracted in a domain T, with the set of abstract states $\mathcal{T} = \mathcal{V}ar \rightarrow \mathcal{P}(\{Int, Str\})$ and domain operations given as point-wise set union and intersection.

Object Abstraction As already mentioned, we use our dictionary analysis technique to model possible attributes of an object. We use a very simple domain R, where each attribute name is represented using a generalised regular expression [8]. We instantiate the domain C by fixing A as $R(\mathcal{V}ar)$, key abstraction as $K = R(\mathcal{V}ar \cup \{v_k\})$ and value abstraction as $V = T(\mathcal{V}ar \cup \{v_v\})$.

Example 3.1 We demonstrate the analysis on the code fragment shown in Figure 6(b). We write attribute accesses in a python-like style, however they are

Project	Size	# meth.	off	on	# inv.
ImageJ	84	4 372	2:08	2:29	265
Apache Commons Math	42	2 896	1:53	1:57	176
MidpSSH	12	561	0:10	0:11	56
Berkeley DB	103	6 196	5:08	5:12	45

Table 1
Array analysis statistics on open-source projects

in fact just dictionary accesses. And so `obj.x` is equivalent to `obj['x']` and `setattr(obj,v,u)` can be written as `obj[v]` $\leftarrow$ `u`.

In the abstract segments we show only the abstract values of the artificial variables v_k and v_v. In the first loop iteration, the scalar at is abstracted as $a(at) = b$. Thus, after setting the attribute in line 3, the object is modelled as $d_3(obj) = \{(b, \{Int\})\}$ and $i_3(obj) = \{(\neg b, \mathsf{True})\}$. In the next loop iteration at is widened to $a(at) = bc^*$. Since bc^* is not a singleton, the `setattr` in line 3 results in a weak update and after the loop we get $d_5(obj) = \{(bc^*, \{Int\})\}$ and $i_5(obj) = \{(\neg b, \mathsf{True})\}$. After the (strong) update in line 7, `obj` is modelled as $d_7(obj) = \{(bc^*, \{Int\}), (x, \{Int\})\}$ and $i_7(obj) = \{(\neg b \wedge \neg x, \mathsf{True})\}$. Similarly, after the assignment in the second branch we get:

$$d_9(obj) = \{(bc^*, \{Int\}), (x, \{Str\})\} \quad i_9(obj) = \{(\neg b \wedge \neg x, \mathsf{True})\} .$$

Thus, joining the states from lines 7 and 9 gives

$$d_{10}(obj) = \{(bc^*, \{Int\}), (x, \{Int, Str\})\} \quad i_{10}(obj) = \{(\neg b \wedge \neg x, \mathsf{True})\} .$$

We can now prove that the attribute usage in line 11 is correct. The attribute b must be present in obj and it is an integer. The access `obj.bcc` is detected as unsafe (possible `AttributeError`). The analysis captured that obj may contain any attribute bc^*, each of type Int, but bcc (or any bc^+) does not need to be present in obj. The last instruction is signalled as unsafe, as `obj.x` does not need to be an integer at this program point (possible `TypeError`).

4 Experiment

We have implemented our technique in a prototype analyser for Java source code. The technique was used to array analysis and was instantiated with the domain of Pentagons [12]. We have measured the performance impact of our technique and the number of discovered non-trivial array invariants (i.e. more precise than 'arbitrary value may be at arbitrary index'). The experiment is summarised in Table 1. For each analysed project we report the size (in kilo lines of code), number of methods, time (in minutes) with array analysis turned off and turned on as well as number of detected invariants. Our results are comparable to those reported for FunArray [4].

5 Related Work

The simplest approaches to array content analysis are array smashing and array expansion [1]. Gopan et al. have proposed a technique, in which array elements are grouped depending on the relation between their indices and scalar variables (elements at indices smaller, equal to and greater than value of some variable) [10]. Halbwachs and Péron [11] use a similar idea to automatically discover relations on elements of an array or even of distinct arrays. Cousot et al. have proposed a technique, in which an array is segmented using simple expressions over scalar variables [4]. Dillig et al. have introduced a notion of fluid updates that relax the dichotomy between strong and weak updates [5]. Using uninterpreted functions, this technique can be adapted for arbitrary containers [6], but still it is not possible to express any non-trivial properties of non-numerical keys (e.g. that a key matches a regular expression, like in our example). Other approaches employ techniques such as predicate abstraction [7] or counter-example guided abstraction refinement [14].

6 Conclusion

We have developed a technique for dictionary and array content analysis. The precision/cost of the analysis depends on the choice of abstractions of scalars as well as dictionary keys and values. We have shown examples, in which our approach was applied to analysis of arrays and string-keyed dictionaries. The solution was experimentally evaluated and gave very promising results.

References

[1] Blanchet, B., P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, *Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software*, in: *The essence of computation*, Springer-Verlag, 2002 pp. 85–108.

[2] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *POPL '77*, pp. 238–252.

[3] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *POPL '79*, 1979, pp. 269–282.

[4] Cousot, P., R. Cousot and F. Logozzo, *A parametric segmentation functor for fully automatic and scalable array content analysis*, in: *POPL'11* (2011), pp. 105–118.

[5] Dillig, I., T. Dillig and A. Aiken, *Fluid updates: Beyond strong vs. weak updates*, in: *ESOP*, 2010, pp. 246–266.

[6] Dillig, I., T. Dillig and A. Aiken, *Precise reasoning for programs using containers*, in: *POPL'11* (2011), pp. 187–200.

[7] Flanagan, C. and S. Qadeer, *Predicate abstraction for software verification*, in: *POPL '02* (2002), pp. 191–202.

[8] Fulara, J., "Abstract Analysis of Numerical and Container Variables," Ph.D. thesis, University of Warsaw (2012).

[9] Fulara, J., K. Durnoga, K. Jakubczyk and A. Schubert, *Relational abstract domain of weighted hexagons*, Electron. Notes Theor. Comput. Sci. **267** (2010), pp. 59–72.

[10] Gopan, D., T. Reps and M. Sagiv, *A framework for numeric analysis of array operations*, in: *POPL '05* (2005), pp. 338–350.

[11] Halbwachs, N. and M. Péron, *Discovering properties about arrays in simple programs*, in: *PLDI*, 2008, pp. 339–348.

[12] Logozzo, F. and M. Fähndrich, *Pentagons: a weakly relational abstract domain for the efficient validation of array accesses*, in: *SAC '08*, pp. 184–188.

[13] Miné, A., *The octagon abstract domain*, Higher Order Symbol. Comput. **19** (2006), pp. 31–100.

[14] Seghir, M. N., A. Podelski and T. Wies, *Abstraction refinement for quantified array assertions*, in: *SAS'09* (2009), pp. 3–18.

[15] Simon, A., A. King and J. M. Howe, *Two variables per linear inequality as an abstract domain*, in: *LOPSTR'02* (2003), pp. 71–89.