Generic Abstraction of Dictionaries and Arrays

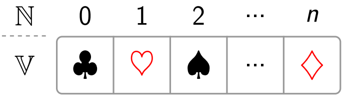
Jędrzej Fulara

University of Warsaw

September 10, 2012

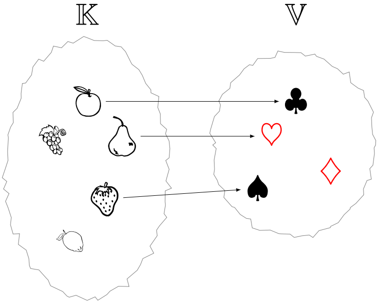
Arrays

- Data structure indexed by an initial range of natural numbers
- Fixed size
- Multiple techniques for static analysis of array content



Dictionaries

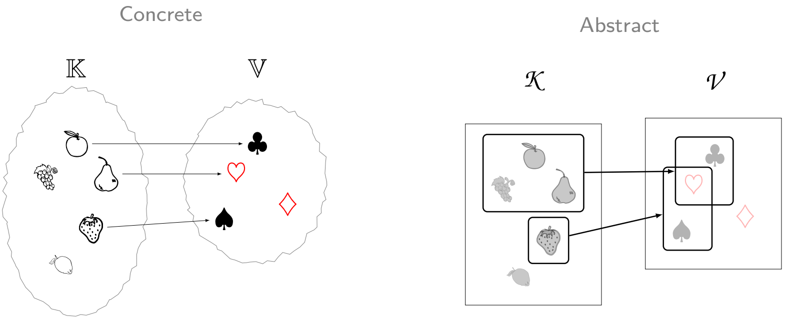
- Arbitrary, possibly non-numerical keys $\mathbb{K}$
- Variable size
- Not handled by static analysis



Objective

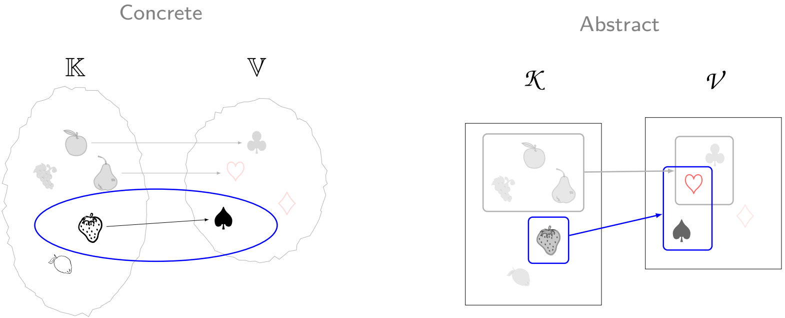
- Abstract Interpretation based technique for analysis of dictionary content
- Fully customisable
  - parametrisable by abstraction of dictionary keys...
  - and dictionary values
- Applicable to analysis of dictionaries and arrays

Dictionary Abstraction



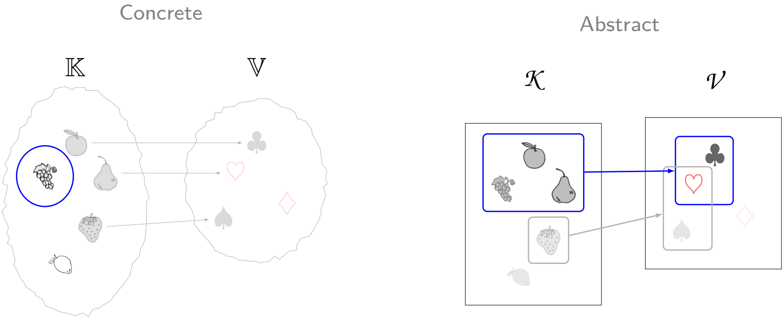
- $\mathbb{K}$, $\mathbb{V}$ - sets of concrete keys and dictionary values
- $\langle \mathcal{K}, \sqcap_k, \sqcup_k, \alpha_k, \gamma_k \rangle$ and $\langle \mathcal{V}, \sqcap_v, \sqcup_v, \alpha_v, \gamma_v \rangle$ - abstract domains for key and dictionary value abstractions
- Abstract dictionary $d \in \mathcal{D}$ as a finite set of pairs $(k, v) \in \mathcal{K} \times \mathcal{V}$ (*abstract segments*)

Dictionary Abstraction



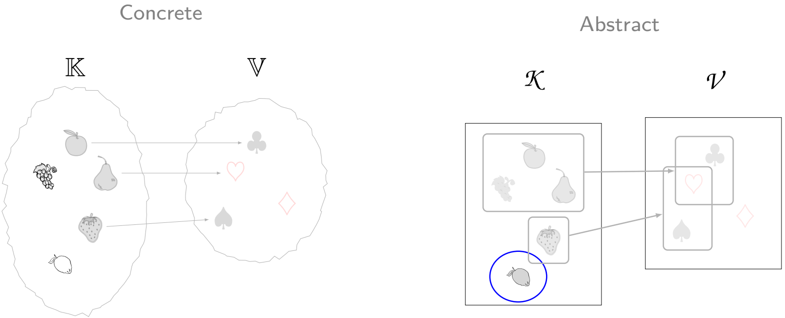
- the abstraction is over-approximating:
  - for $(k, v) \in d$, v over-approximates the set of values of elements at keys abstracted by k,
  - a concrete key not represented by any abstract one cannot be initialised

Dictionary Abstraction



- the abstraction is over-approximating:
  - for $(k, v) \in d$, v over-approximates the set of values of elements at keys abstracted by k,
  - a concrete key not represented by any abstract one cannot be initialised

Dictionary Abstraction



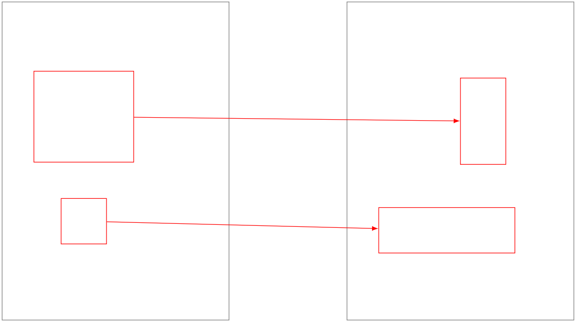
- the abstraction is over-approximating:
  - for $(k, v) \in d$, v over-approximates the set of values of elements at keys abstracted by k,
  - a concrete key not represented by any abstract one cannot be initialised

Dictionary Abstraction

- no two abstract keys may overlap:
  - each concrete element is represented by just one abstract segment
- abstract keys and abstract values are non-empty (i.e. $k \neq \bot_k$ and $v \neq \bot_v$)
  - $(\bot_k, v)$ would describe an empty segment
  - $(k, \bot_v)$ would describe uninitialised elements

Lattice Operations - meet

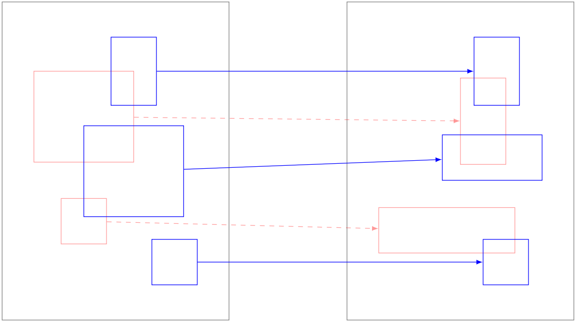
Point-wise meet of overlapping segments:



For $a \in \mathcal{D}$

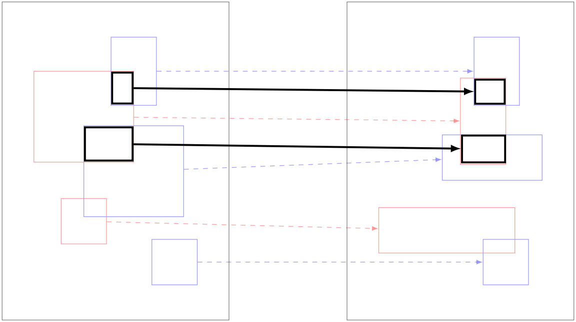
Lattice Operations - meet

Point-wise meet of overlapping segments:



For $a \in \mathcal{D}$ and $b \in \mathcal{D}$:

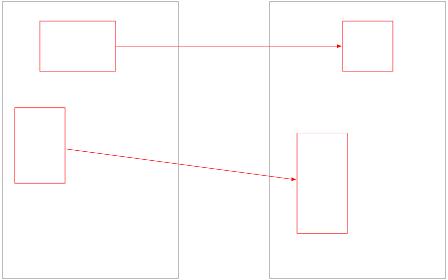
Point-wise meet of overlapping segments:



For $a \in \mathcal{D}$ and $b \in \mathcal{D}$:

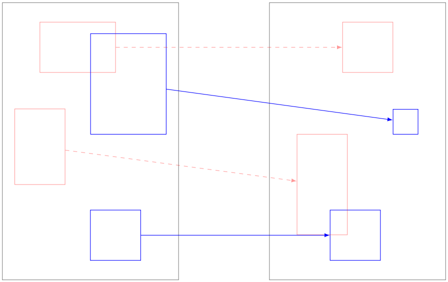
$$a \sqcap_d b \triangleq \{(k_a \sqcap_k k_b, v_a \sqcap_v v_b) \mid (k_a, v_a) \in a, (k_b, v_b) \in b, k_a \sqcap_k k_b \neq \bot_k, v_a \sqcap_v v_b \neq \bot_v\}$$

Lattice Operations - Join



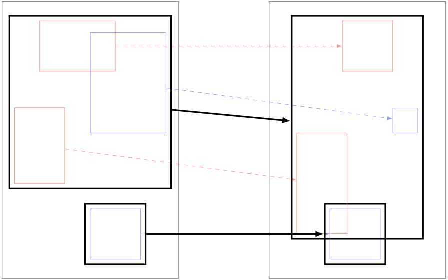
- take the union of the two operands,
- transform it so that no two abstract keys overlap

Lattice Operations - Join



- take the union of the two operands,
- transform it so that no two abstract keys overlap

Lattice Operations - Join



- take the union of the two operands,
- transform it so that no two abstract keys overlap

Disjoint Partitions

Let $A = \langle \mathcal{A}, \sqcap_a, \sqcup_a \rangle$ be a complete lattice and let $S \subseteq \mathcal{A}$

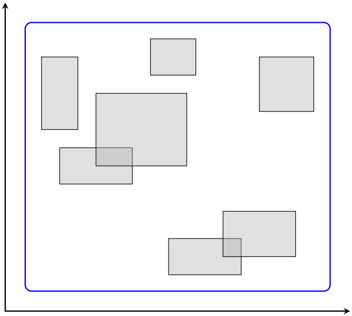
Definition

$\mathcal{X} = \{X_1, \ldots, X_k\}$, where $X_i \subseteq S$, 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 $i \neq j$, $(\bigsqcup_a X_i) \sqcap_a (\bigsqcup_a X_j) = \bot_a$.

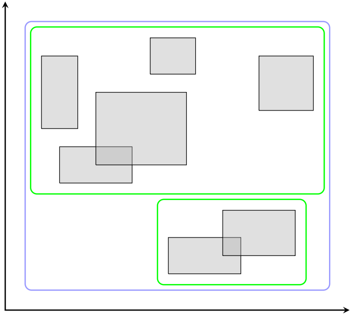
Disjoint Paritititions

- disjoint partition always exists (e.g. $\mathcal{X} = \{S\}$),



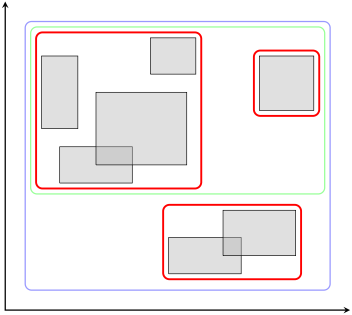
Disjoint Parititions

- disjoint partition always exists (e.g. $\mathcal{X} = \{S\}$),
- there may be many of them



Disjoint Paritititions

- disjoint partition always exists (e.g. $\mathcal{X} = \{S\}$),
- there may be many of them
- the least one is unique



To compute the join $a \sqcup_d b$:

- find the least disjoint partition $\mathcal{K}$ of keys in $a \cup b$
- for $K \in \mathcal{K}$, take the join of segments from $a \cup b$ with keys in K

Set $\mathcal{D}$ forms a lattice under $\sqcap_d$ and $\sqcup_d$.

Relational Analysis

Goal:

- express relations between dictionary keys/values and scalar variables $\mathcal{Var}$

Solution:

- artificial *key variable* v_k and *value-tracking variable* v_v
- key abstraction K over $\mathcal{Var} \cup \{v_k\}$
- dictionary values abstraction V over $\mathcal{Var} \cup \{v_v\}$

Initialisation Analysis

Problem:

- our dictionary abstraction $D(K, V)$ is over-approximating,
- no information which dictionary elements must be initialised

Solution:

- separate initialisation analysis using $D(K, Bool)$,
- over-approximates set of uninitialised keys
- segment $(k, True)$: keys abstracted by k may be uninitialised

Abstract Domain

Parameters:

- abstraction of scalars in a domain $A(\mathcal{Var})$
- key abstraction $K(\mathcal{Var} \cup \{v_k\})$
- value abstraction $V(\mathcal{Var} \cup \{v_v\})$

The domain:

- Abstract state:
  $A \times (\mathcal{Var}_d \to D(K, v)) \times (\mathcal{Var}_d \to D(K, Bool))$
- meet and join given point-wise,
- "lazy" widening

Abstract states denoted as triples (a, d, i)

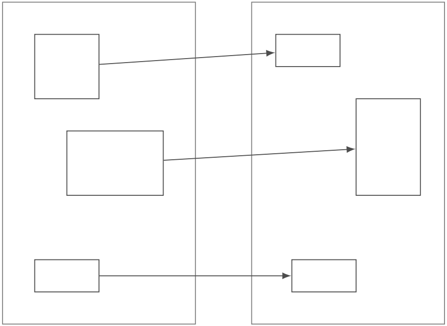
Concretisation

Informally:

- a key n in a dictionary T may be uninitialised, if there is a segment $(k, \mathit{True}) \in i(T)$, such that n is abstracted by k,
- $T[n]$ may have value e, if $(k, v) \in d(T)$ and n and e are abstracted by k and v, respectively

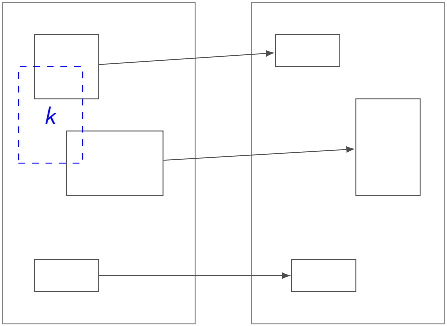
Dictionary Read

- read access $x \leftarrow T[e]$,



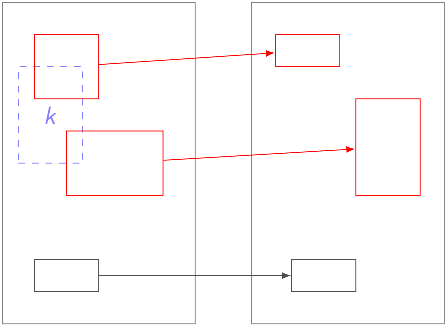
Dictionary Read

- read access $x \leftarrow T[e]$,
- compute abstract key k representing e



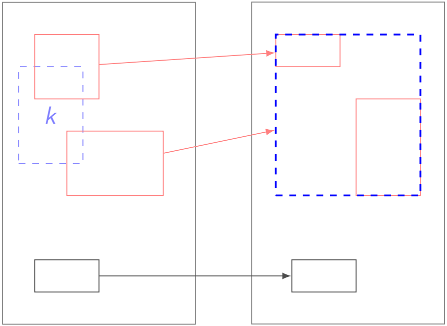
Dictionary Read

- read access $x \leftarrow T[e]$,
- compute abstract key k representing e
- find in the dictionary abstract keys intersecting with k



Dictionary Read

- read access $x \leftarrow T[e]$,
- compute abstract key k representing e
- find in the dictionary abstract keys intersecting with k
- compute join of corresponding abstract values



Dictionary Updates

Consider an update $T[x] \leftarrow 42$.

- Assume that the analysis of scalars captured $x \in [0, 5]$
  - Any of $T[0] \ldots T[5]$ may be equal to 42 or to its old value
  - None of them must be equal to 42
  - The old values of $T[0] \ldots T[5]$ cannot be purged

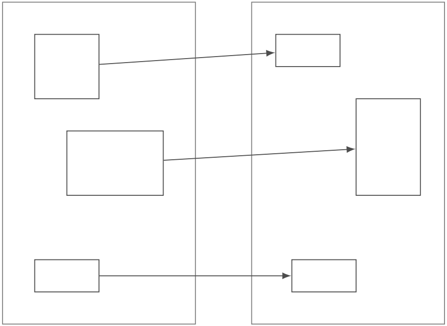
  This is called a *weak update*

- Assume that $x \in [3, 3]$
  - $T[3]$ must be equal to 42
  - Its old value can be forgotten

  This is called a *strong update*

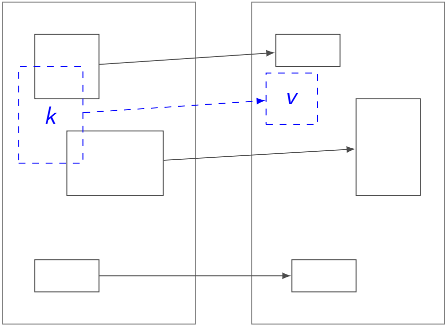
Weak Update

- update $T[x] \leftarrow y$,



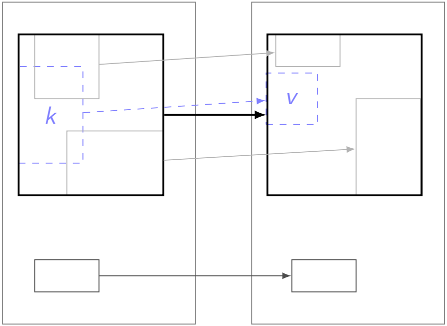
Weak Update

- update $T[x] \leftarrow y$,
- compute abstract segment (k, v) representing (x, y)



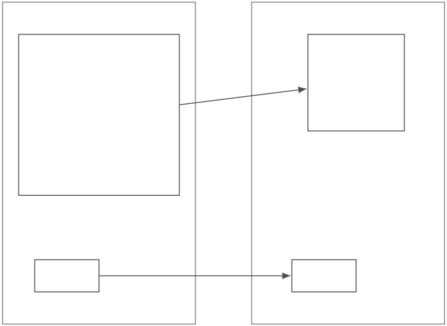
Weak Update

- update $T[x] \leftarrow y$,
- compute abstract segment (k, v) representing (x, y)
- smash segments with overlapping keys



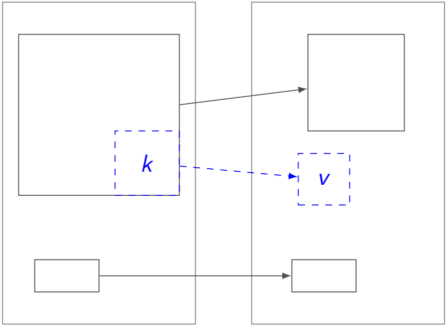
Strong Update

- update $T[x] \leftarrow y$,



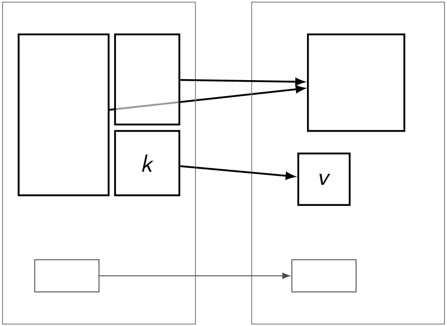
Strong Update

- update $T[x] \leftarrow y$,
- compute abstract segment (k, v) representing (x, y)



Strong Update

- update $T[x] \leftarrow y$,
- compute abstract segment (k, v) representing (x, y)
- cut it out from the existing segments



Examples [1/3]

- scalars, key and values abstraction using upper bounds
- each variable mapped to the set of variables greater or equal to it

```
procedure Partition(T, x)
  l = 0; r = T.length - 1;
  while l < r do
    if T[l] <= x then
      l = l + 1
    else if T[r] >= x then
      r = r - 1
    else
      y = T[r]; T[r] = T[l];
      T[l] = y;
    end if
  end while
```

Found invariant: $\forall_{0 \leq m < l} T[m] \leq x$ and $\forall_{l < n < T.length} x \leq T[n]$.

- scalars and keys: abstraction using product: *Upper Bounds* $\times$ *Parity*
- dictionary values: abstraction using intervals

```
j = 0; T = new array[n];
while j < n
  if j % 2 = 0 then
    T[j] = 1
  else
    T[j] = 0
  end if
end while
```

All even elements equal to 1, all odd ones equal to 0.

Examples [3/3]

Dynamic programming languages: object ≡ string-keyed dictionary

- keys: abstraction using regular expressions
- values: abstraction by type (Int, String,...)
- detecting missing attribute errors
- detecting type errors

```
at = "b"
repeat
  setattr(obj, at, 6);
  at = at + "c";
until random() = False;
if random() = True then
  obj.x = 5
else
  obj.x = "text"
end if
x = obj.b - 1;
y = obj.bcc - 1;
z = obj.x - 1;
```

Summary

- new abstract domain for analysis of arbitrary dictionaries
- fully customisable
- starting point to static analysis of dynamic languages