Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Usage Analysis of Data Stored in Map Data Structures

Bachelor's Thesis

Lowis Engel

Supervised by Dr. Caterina Urban, Jérôme Dohrau
and Prof. Dr. Peter Müller

Department of Computer Science, ETH Zurich

August 28, 2018

Abstract

With the raise of data science and similar fields there is a fast growing number of programs working with large sets of data. As these programs get more complicated and the data to analyse increases, it may get harder to detect non-fatal errors in the computation. Especially it may not be noticed if some of the input data is not used. Here we use a strong definition of data usage: We only declare data as used if there is a dependency between the content of the data and the outcome of the program. Using the concept of abstract interpretation we extended existing analyses to be able to analyse this usage property on map data structures. Our approach generically applies to all such data structures containing key-value pairs.
As a preliminary step we analyse the possible values of scalar variables and map data structures. We are able to divide the key space of a dictionary, and thus can infer different possible values for different keys. For this analysis and our abstraction of map data structures we build on previous work by Fulara [3].
Using the results of this first analysis we then analyse the usage of map data structures with two different abstractions. With these analyses we are able to infer sets of keys whose corresponding values are unused.
All these analyses are generic in their abstraction of keys, values and scalar variables. Hence, they can be adjusted to various needs of precision and data types.

Contents

1 Introduction

With the ever increasing amount of available digital data, analysing it is getting more and more relevant. This led to the rise of data science for decision making in many fields. However, as the dependence on the decisions of these applications grows, errors can have increasingly severe consequences. Hence, it is important to ensure correctness of such programs. Therefore the goal of the Lyra project[1] is to provide a framework for statical analyses of such programs to infer properties of their behaviour which are useful to detect errors.

Non-fatal programming errors, where there is no clear indication that something went wrong, can be particularly dangerous. One type of these errors, which is relevant especially in data science, is entirely unused input data. Programs affected by such faults may still lead to plausible results, especially if a program is operating on a large possibly unknown data set and is performing a long pipeline of operations. Here we define input data to be *used* if there is a dependency between the input and the outcome of a program (directly or indirectly via conditions or assignments). This means it is used, if the outcome of the program can be influenced by different concrete input values.

```
scores: Dict[int, int] = dictinput()     # id -> score
previous_ge_10: int = int(input())       # previous results

score_occurrences = defaultdict(int)     # initialized to 0
for a, b in scores.items():
    if a < 100:      # 'early adopter'
        weight: int = 3
    else:
        weight: int = 1
        score_occurrences[a] += weight    # BUG A: should be
                            indexed by b & BUG B: wrong indentation


scores_gt_10: int = 0
if previous_ge_10 > 0:
    scores_ge_10 = previous_ge_10   # BUG C: scores_ge_10
                    instead of scores_gt_10

for k in score_occurrences.keys():
    if k > 10:
        scores_gt_10 += score_occurrences[k]

print(scores_gt_10)
```

Example 1: Motivating Example

Consider the Python program above as an example for mistakenly unused input data. Here a dictionary is given as input. One can think of the input dictionary being a table which maps user identifiers to a score. The data could for example come from a user study for a new product, which the users rated with the score. The program aggregates this data by first translating the dictionary to another dictionary (‘`score_occurrences`’), which should map the different scores to how often

[1] `http://www.pm.inf.ethz.ch/research/lyra.html`

they occur in the data. Here the score of persons with an identifier lower than 100 are counted two times more than the scores of other persons. One could imagine them being the first persons who tried the product. Hence, they may have used it for the longest time and one may want to value their opinion higher. In the end, the counts for scores that are higher than 10 should get summed up and returned. Before that a previous result from '`previous_ge_10`' should be added.
There are three errors in the program: Because of 'Bug A' it does not count the number of occurrences of a score, but rather the occurrences of the identifiers (which always occur only once). 'Bug B' causes the summation on line 10 to be executed only for persons with an identifier greater or equal to 100. Finally because of the spelling mistake in 'Bug C' the previous result will never get added to '`scores_gt_10`'. Since we sum up the data again in the end, these bugs may not be noticed by looking at the output.

Naturally usage information propagates backwards. The only output of the program '`scores_gt_10`'. Therefore in the beginning only this variable is used. This causes '`score_occurrences[k]`' in line 18 to be used and finally '`a`' and '`weight`' in line 10 get used. One can see that the values of the '`scores`' dictionary are never used, since due to 'Bug A' only the keys '`a`' and not the values '`b`' are used inside the loop. Without 'Bug A' still not all values of '`scores`' are used, since only the values with a key $\mathtt{a} \geq 100$ are used because of 'Bug B'. Therefore we can detect both bugs with a sufficiently precise usage analysis. '`previous_gt_10`' is never used, since it gets assigned to the wrong (unused) variable due to 'Bug C'.

The goal of this thesis is to design usage analyses that can analyse map data structures, i.e. data structures consisting of key-value pairs, and to implement them for Python's dictionary[2] and list[3] types. Here we treat lists as a special type special type of map data structures. Hence, we abstract them in the same way as dictionaries and ignore the fact that they have keys that always range from zero to one less than the length of the list.

Using the concept of *abstract interpretation* [1] we define abstract semantics to reason about this usage property in a static context. Based on existing analyses in Lyra we abstract input data usage with two different abstractions. The *strongly live* abstraction, introduced in [4], is an extension of the live variable analysis used by compilers and covers information flow through assignment statements. The usage abstraction deals with implicit usage by conditionals more precisely. It is defined in [5, 7], where also the strongly live abstraction is evaluated. Both abstractions overapproximate the usage property we defined above, such that we can report data that is *definitely* unused.

We implement our analyses in Python for programs written in Python, one of the most popular programming languages in data science. Usage analyses for simple programs only containing scalar variables, meaning variables that only hold one value at a time, and Python's lists were already implemented in Lyra. Lists are only abstracted by a summary abstraction in this implementation, which means a list is abstracted as used if any of its elements is possibly used. In [7] Simon Wehrli has already described a more precise analysis of lists by reasoning about the usage of list segments.

We propose more general domains which work for all map data structures and different abstractions of keys and values. Similarly to Wehrli's work these domains

[2]`https://docs.python.org/3/library/stdtypes.html#dict`
[3]`https://docs.python.org/3/library/stdtypes.html#list`

divide the key space to abstract the usage of parts of a map data structure. For this we need to be able to determine the values of which keys are used. Hence, similar to the interval analysis that needs to be run for lists, we need to run a preliminary analysis to determine the values of key expressions.

Outline. In Section 2 we will present some definitions that form the basis of this thesis. In Section 3 we will show how we abstract map data structures and define a preliminary analysis to analyse the values of scalar variables and contents of map data structures. In Section 4 we describe the two usage analyses for map data structures using different levels of precision. Some implementation details and an evaluation can be found in Section 5. Finally, Section 6 concludes this thesis.

2 Background

2.1 Static Analysis by Abstract Interpretation

Abstract interpretation [1] is a unified way of reasoning about properties of program behaviour in a mathematical way. A formal introduction can be found in [6]. Intuitively, static analysis by abstract interpretation iterates over all statements of a program producing an *abstract state* before or after every statement until a fixpoint is reached. The analysis is performed in a static context, and thus all possible program behaviours need to be reflected. Therefore we can often only approximate the properties we want to analyse.

To run an analysis by abstract interpretation we need to define an *abstract domain* which defines the possible abstract states and the operations to compute them. The domain consists of:

- **Elements.** The elements of the abstract domain are the possible abstract states. They abstract (and approximate) the potential concrete states at a certain point in the program in terms of our desired properties.
- **Order.** The elements should belong to a complete lattice whose partial order relation $\sqsubseteq$ captures the *precision* of an approximation, i.e. how many concrete states it represents. This means for a relation $a \sqsubseteq b$, a is said to be more precise than b, since b abstracts at least the concrete states abstracted by a. Therefore b can also be called an upper bound on a.
- **Bottom and Top.** The bottom and top elements are the least and greatest elements of the lattice respectively in terms of the relation $\sqsubseteq$. They get denoted by the symbols $\bot$ and $\top$, respectively.
- **Join.** When two results of different iterations of the analysis or from different execution paths in the program reach the same point, we want to combine them to one abstraction covering both elements. This is similar to a logical 'or', meaning the concrete state may be abstracted by either of the two states. Hence, the join operation $\sqcup$ computes (an overapproximation of) the least upper bound of the given states with respect to the order $\sqsubseteq$.
- **Meet.** The meet $\sqcap$ is the opposite of the join. It computes (an overapproximation of) the greatest lower bound of multiple abstract states. It can be used if we know that the concrete state must be part of multiple abstractions. This means, we want to compute a new state which is similar to a logical 'and' of the information of those abstractions.

- **Widening.** If the program we analyse contains loops, the analysis may iterate infinitely or infeasibly long, since the abstraction may only slowly increase in each loop iteration. Therefore we need the widening operation $\triangledown$, which approximates the join after a constant number of iterations with a domain element that is sufficiently imprecise to ensure convergence to a fixpoint. The widening operation will be definitely required if the domain's order has an infinite height. By infinite height we mean that for some or all elements there exist infinitely many elements that are larger in terms of $\sqsubseteq$. Clearly, such a domain leads to an infinite iteration for some programs.
- **Abstract Transfer Function.** We need to transform our abstractions to reflect the effects of the statements of a given program while iterating over them. For that we need to define *abstract transformers* that compute a new abstract state from the state before for each (relevant) statement. We denote them by these brackets: $[\![\cdot]\!]$.

Such a static analysis can be performed in two different directions. If we iterate over the statements in the same order as they would be executed, the analysis will be called a *forward analysis.* If we iterate in the opposite direction, we will call it a *backward analysis.*

Interval Domain The interval domain [1] is a well-known simple numerical domain. It represents each variable by an interval approximating its possible values. The bound of an interval may possibly be positive or negative infinity.
We used the interval domain to evaluate our analyses.

2.2 Abstractions of Usage

We call all elements whose usage we analyse *data items.* For our analyses these are scalar variables and map data structures, i.e. lists and dictionaries, or parts of them. As stated before, we say a data item is used if different values stored in it cause different outcomes of a program. Otherwise we say it is unused. Formal definitions of the usage property are given in [7, 5].
Here we will describe the two previously mentioned abstractions of this property, namely the strongly live abstraction and the usage abstraction. Both of them map data items to an abstraction of the usage property, which we call *abstract property.* As a first step we extended existing implementations of analyses with these abstractions to deal with dictionaries using a *summary abstraction* of containers. By summary abstraction we mean that we map the whole map data structure (variable) to one abstract property which is the upper bound of the concrete usage of all elements in the data structure. This means if any element of the map data structure is used, we abstract that by saying the whole data structure is used.

2.2.1 Strongly Live Abstraction

The *strongly live* (variable) abstraction maps each data item to *Live* (*strongly live*) or *Dead* (sometimes called *faint*). A data item is considered strongly live if it occurs on the right-hand side of an assignment to a strongly live data items, or if it appears in any other statement (including conditionals). Otherwise it is considered dead. This means strongly liveness propagates through a chain of assignments, transferring the property from the left-hand side to the right-hand side. Definitions of abstract transformers are given in [5].

This abstraction overapproximates the usage property. Hence, *Live* means the variable may be used and *Dead* means it is definitely unused.
For the implementation analysing map data structures we need to adapt the definition of strongly live a little bit to still overapproximate usage for assignments to subscriptions like '`d[k]`'. Because subscripts like '`k`' are used to select the element of a list or dictionary which is accessed, different values may change the outcome of the program by selecting a different element. Therefore data items occurring in subscripts are used and need to be marked as *Live*, even if they occur on the left-hand side of an assignment.

If we run this analysis on Example 1, the '`scores`' dictionary will not be marked as *Live*. Therefore we could detect that there is an error. 'Bug B' can not be detected by this analysis, since the error causes only a part of '`scores`' to be used (if 'Bug A' is resolved), which our summary abstraction cannot reflect. Hence, in this case the analysis reports that the whole dictionary is *Live*. Moreover, this analysis cannot notice 'Bug C' due to the imprecision of the strongly live abstraction: '`previous_ge_10`' will always be marked as *Live*, because it appears in the condition in line 13 and the analysis does not take into account that there is no usage in this branch.

2.2.2 Usage Abstraction

As we have seen, the strongly live abstraction is not precise enough to detect errors like 'Bug C'. Therefore Urban, Müller [5] and Wehrli [7] introduced a more precise abstraction. This abstraction takes nesting levels of loops and conditionals into consideration and can therefore tell whether the usage information changed inside a branch. This helps us to mark data items appearing in conditions only as used if they have an effect on the outcome of the program (according to our current abstraction).
There are four levels of abstraction a variable (or data item) can map to in this abstraction. The abstract properties U (*$\underline{u}$sed*) and N (*$\underline{n}$ot used*) are similar to *Live* and *Dead* of the strongly live abstraction. They mean that a data item may be used or that it is definitely not used *at the current nesting level*. Additionally, the property B (*$\underline{b}$elow*) is used to denote that a data item may be used (i.e. is U) somewhere below in a lower nesting level, but has not (yet) been used at the current nesting level. Finally, W (*over$\underline{w}$ritten*) indicates that the data item may be used below and that it has been overwritten in the current nesting level.
To keep track of the state of outer nesting levels the abstractions of all current levels are maintained in a stack of abstract usage states which is formally described in [7]. When we enter a deeper nesting level, we push a new abstraction for it on the stack. This new abstraction is computed by the $\textsc{inc}(u)$ function which marks every data item that was used in the outer scope (U) as used below (B) and every data item that was over written in the outer scope (W) as not used (N). When we exit an inner nesting level again, we remove (pop) its abstract state from the stack and update the abstraction of the outer nesting level (i.e. the next abstraction on the stack) with this information using the $\textsc{dec}(u^{(inner)}, u^{(outer)})$ function. This function updates data items to be used (U) or overwritten (W) if they were used or overwritten in the inner scope. Other data items keep their abstract property from the outer scope. In other words, we restore the abstract properties that were changed by the $\textsc{inc}$ function.

The analysis results for Example 1 using summary abstraction are quite similar to the results of the strongly live analysis. The difference is that this abstraction is able to detect 'Bug C', since no variable becomes W or U in the branch of the condition

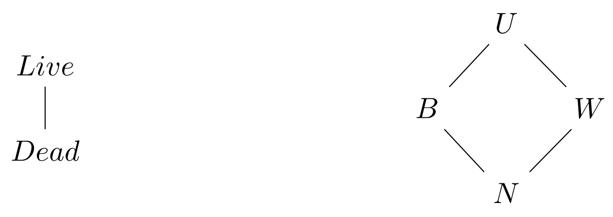

Figure 1: Hasse diagrams for the abstract usage properties

in line 13 and because of this ‘`previous_ge_10`’ appearing in the condition does not get marked as used.

2.3 Simplified Python Language

For our formal definitions we use a simplified version of the Python language from which one can easily derive more complex language constructs. It is based on the language definition of ‘SimplePython’ in [7]. The syntax is defined as follows:

Constants:

$$\begin{array}{rll}
i & \in \mathbb{Z} & \text{integer constants} \\
s & \in \{\mathrm{A} \,|\, ... \,|\, \mathrm{Z} \,|\, \mathrm{a} \,|\, ... \,|\, \mathrm{z}\}^* & \text{string constants} \\
c & \in \mathbb{Z} \cup \{\mathrm{A} \,|\, ... \,|\, \mathrm{Z} \,|\, \mathrm{a} \,|\, ... \,|\, \mathrm{z}\}^* & \text{general constants}
\end{array}$$

Variables:

$$\begin{array}{rll}
x, y, z & \in \mathbb{V}ar_{scalar} & \text{scalar variables} \\
d, d_1, d_2 & \in \mathbb{V}ar_{dict} & \text{dictionary variables} \\
l, l_1, l_2 & \in \mathbb{V}ar_{list} & \text{list variable} \\
dl, dl_1, dl_2 & \in \mathbb{V}ar_{map} := \mathbb{V}ar_{dict} \cup \mathbb{V}ar_{list} & \text{map variable} \\
v, v_1, v_2 & \in \mathbb{V}ar := \mathbb{V}ar_{scalar} \cup Var_{map} & \text{general variables}
\end{array}$$

Scalar expressions:

$s \in \mathbb{E}xpr_{scalar} ::=$ *Any Python expression only containing scalar variables (that the scalar domain allows) including ‘*`input()`*’*

Scalar boolean expressions:

$sb \in \mathbb{B}expr_{scalar} \subset \mathbb{E}xpr_{scalar} ::=$ *Any Python boolean expression only containing scalar variables (that the scalar domain allows)*

Membership boolean expressions:

$$\begin{array}{rll}
mb \in \mathbb{B}expr_{dict} ::= & y \ \texttt{in} \ l & \text{list membership} \\
| & x \ \texttt{in} \ d_1\texttt{.keys()} & \text{key membership} \\
| & y \ \texttt{in} \ d_1\texttt{.values()} & \text{value membership} \\
| & x, y \ \texttt{in} \ d_1\texttt{.items()} & \text{item membership} \\
| & y \ \texttt{not in} \ l & \\
| & x \ \texttt{not in} \ d_1\texttt{.keys()} & \\
| & y \ \texttt{not in} \ d_1\texttt{.values()} & \\
| & x, y \ \texttt{not in} \ d_1\texttt{.items()} &
\end{array}$$

Boolean expressions:

$$b, b_1, b_2 \in \mathbb{B}expr ::= sb \mid mb$$

Expressions:

$$e \in \mathbb{E}xpr ::= s \mid mb$$

Map data structure expressions:

$me \in \mathbb{M}expr ::=$	`listinput()`	list input
	$\mid$ `dictinput()`	dictionary input
	$\mid$ `[]`	new empty list
	$\mid$ `{}`	new empty dictionary
	$\mid dl1$	

Statements:

$stmt \in \mathbb{S}tmts ::=$	`pass`	no effect
	$\mid x = s$	scalar assignment
	$\mid dl = me$	map assignment
	$\mid dl[x] = y$	subscription write
	$\mid y = dl[x]$	subscription read
	$\mid$ `print`(e)	output
	$\mid$ `if` b: $stmt_1$ `else`: $stmt_2$	conditional
	$\mid$ `for` mb: $stmt_1$	for-loop
	$\mid stmt_1$ <newline> $stmt_2$	statement sequence

We assume that the types of all variables are either provided by the user via type annotations or can be inferred by a type checker. Obviously all statements and expressions should follow the typing rules of Python.

Since our analyses are intra-procedural, we use the various input functions to simulate actual parameters given to a function or inputs from I/O. Therefore we included the conceptual '`dictinput()`' and '`listinput()`' functions, which are not part of the built-in functions of Python. They can also be seen as a placeholder for special library functions to read in files, e.g. tables/databases as dictionaries[4].

For the rest of this report we will use the names introduced above, when we show lines of code with placeholder variables.
We also introduce an auxiliary function $vars(s)$ to compute all scalar variables in a scalar expression s.

Remark: The Lyra framework deals with '**for**'-loops not by iterating over them with one value at a time, but by computing an abstraction of all possible values of the loop variable(s). This may introduce some imprecision in some cases. We denote the abstract transformers for the loop header by $[\![for \cdot \texttt{in} \cdot]\!]$. For the exit condition we write $[\![for \cdot \texttt{not in} \cdot]\!]$.
Also not that the abstract transformers for the '**pass**' statement are always defined to not change the abstract state. Furthermore, the abstract transformers for the 'statement sequence' are defined by applying the abstract transformer for the first statement and then applying the abstract transformer for the second statement on that result.

[4]cf. `https://docs.python.org/3/library/csv.html#csv.DictReader`

2.4 Restrictions

In general we assume that all programs we analyse are deterministic and single-threaded.

Our domains do not support aliasing. Hence, a map data structure may be aliased by an assignment $dl_1 = dl_2$, but the two variables will be treated as if they would refer to different copies of the same object. This means, when the state for one such map variable is updated, the other one will not be updated simultaneously.

Another restriction is that the presented analyses do not support nested map data structure, i.e. map data structures whose values are also map data structures.

Also we assume that sets of elements do not change, i.e. no elements get added or deleted, while we iterate over it, since even in Python this leads to undefined behaviour.

3 Map Data Structure Content Analysis

As we have seen in Section 2.2 a summary abstraction is often not sufficient, since we want to determine the usage of parts of map data structures. Therefore we need the ability to determine, which parts are possibly represented by a program expression. Especially, we want to be able to evaluate the key expression of a subscription like `d[key]`.
For this reason we run a forward analysis first, that gives us an abstraction of the possible values of variables and the possible values corresponding to an abstracted set of keys for map data structures.
This analysis is mostly based on work by Fulara [3, 2]. Below, we summarize Fulara's analysis together with a description of our extensions to it.

3.1 The Segment Lattice

Map data structures get abstracted by a lattice whose elements are sets of so-called abstract segments. These abstract segments are pairs consisting of an abstract key and an abstract value from some domains $\langle \mathcal{K}, \sqsubseteq_{\mathcal{K}}, \sqcup_{\mathcal{K}}, \sqcap_{\mathcal{K}}, \bot_{\mathcal{K}}, \top_{\mathcal{K}}, \nabla_{\mathcal{K}} \rangle$ and $\langle \mathcal{V}, \sqsubseteq_{\mathcal{V}}, \sqcup_{\mathcal{V}}, \sqcap_{\mathcal{V}}, \bot_{\mathcal{V}}, \top_{\mathcal{V}}, \nabla_{\mathcal{V}} \rangle$ respectively. They represent a set of concrete keys and their corresponding possible concrete values. In the following we will often just write segment if we mean such an abstract segment.

We say that two segments *overlap* if the meet of their keys is not bottom.

Since we want to have a normal form of our representation, Fulara introduces two conditions that a set of segments in the lattice needs to fulfil:

1. The segments in the set should not overlap.
2. The segments must not contain a bottom element (neither as a key nor as a value abstraction).

The first condition means that our abstraction cannot contain different segments abstracting the same concrete keys. By the second condition we do not allow segments which do not abstract any concrete key-value pairs. Also we do not want to include definitely uninitialized values, which could be represented by a bottom value abstraction.
The second condition is easy to check by comparing the key and value abstractions to the bottom element of the corresponding domain. To make an arbitrary segment

set conform with the first condition Fulara introduces a function called *dNorm*, which basically joins all overlapping segments. A formal definition is given in [3].

Lattice Definition. The complete lattice $\langle \mathcal{S}, \sqsubseteq_s, \sqcup_s, \sqcap_s, \bot_s, \top_s, \nabla_s \rangle$ over these segment sets can be formally defined as follows:

$$\begin{aligned}
&a \sqsubseteq_s b :\Leftrightarrow \forall\, (k_a, v_a) \in a.\ \exists\, (k_b, v_b) \in b.\ k_a \sqsubseteq_{\mathcal{K}} k_b \wedge v_a \sqsubseteq_{\mathcal{V}} v_b \\
&\qquad (\textit{i.e. all segments of } a \textit{ are `contained' in a segment of } b) \\
&a \sqcup_s b := dNorm\,(a \cup b) \\
&a \sqcap_s b := \{(k_a \sqcap_{\mathcal{K}} k_b, v_a \sqcap_{\mathcal{V}} v_b) \mid (k_a, v_a) \in a, (k_b, v_b) \in b, k_a \sqcap_{\mathcal{K}} k_b \neq \bot_{\mathcal{K}}, v_a \sqcap_{\mathcal{V}} v_b \neq \bot_{\mathcal{V}}\} \\
&\quad \top_s := \{(\top_{\mathcal{K}}, \top_{\mathcal{V}})\}
\end{aligned}$$

Note that in Fulara's work $\sqsubseteq_s$ is implicitly defined by the relation $a \sqsubseteq_s b \Leftrightarrow a \sqcup_s b = b$.

In contrast to Fulara, who chose the empty set, abstracting the empty data structure, to be the *least element*, we define $\bot_s$ to mean that there is no possible concretisation of the data structure at that point. With this modification our abstraction can distinguish between empty and uninitialized map data structures.

The *widening* operation ∇_s is defined point-wise for overlapping segments, keeping non-overlapping segments from the left operand and applying the *dNorm* function afterwards. If the right operand has a non-overlapping segment, all segments from both operands get joined to one segment with $\top_{\mathcal{K}}$ as the key. In this case the widening is rather imprecise. Fulara describes a more precise, but also more complicated, widening operation in his PhD thesis [2]. However, as we will see the simpler widening is sufficient in most cases, especially if we are careful about the point, when we apply it.

Segment Addition Operators. For assignments we need the possibility to add new concrete key-value-pairs to the abstraction (*weak update*) and to overwrite existing ones (*strong update*). For this purpose we define three different operators, which update a given lattice element a to incorporate a new abstract segment (k', v') abstracting the new key-value pairs. The first two perform weak updates, whereas the last one is used for strong updates.
The last two operators further subdivide the existing segments. Since we cannot create new key abstractions for the subdivision from a generic domain, we require the key domain $\mathcal{K}$ to provide a decomposition function $decomp_{\mathcal{K}} : \mathcal{K} \times \mathcal{K} \to \mathcal{P}(\mathcal{K})$ which returns all 'parts' of the first argument that do not overlap with the second. Formal specifications for this function are given in [3]. If it is not possible to construct such a decomposition that abstracts exactly the same concrete keys as the first argument abstracted before, for some or all possible arguments, a weak update without partitioning (i.e. the first operator) can be performed instead.

The first weak update operator is similar to the join, meaning the segment gets added to the set and then joined with overlapping segments:

$$a \otimes (k', v') := dNorm(a \cup \{(k', v')\})$$

The second, more precise, operator joins the value abstraction of the new segment only with overlapping parts from our old lattice element. It does not join any keys and thus, in contrast to the first operator, the overlapping parts will not be joined

to one segment.
To compute the parts of the new key abstraction k' that do not overlap with any segment in a we define a function that performs a sequence of decompositions:

$$\mathit{Decomp}_{\mathcal{K}}(k, \{w_1, w_2, \dots, w_n\}) := \bigcup_{r \in \mathit{Decomp}_{\mathcal{K}}(k, \{w_2, \dots, w_n\})} \mathit{decomp}_{\mathcal{K}}(r, w_1)$$

If either k' or v' are bottom the operator $\odot$ leaves a unchanged, since we do not allow segments with a key or value abstraction that is bottom and also joining bottom with existing value abstractions would not change them.

$$\begin{aligned} a \odot (k', v') := & \left(a \setminus \{(k, w) \mid (k, w) \in a \text{ and } k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}}\}\right) \\ & \cup \{(m, w) \mid (k, w) \in a,\ k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}} \text{ and } m \in \mathit{decomp}_{\mathcal{K}}(k, k \sqcap_{\mathcal{K}} k')\} \\ & \cup \{(k \sqcap_{\mathcal{K}} k',\ w \sqcup_{\mathcal{V}} v') \mid (k, w) \in a \text{ and } k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}}\} \\ & \cup \{(k'_{rest}, v') \mid k'_{rest} \in \mathit{Decomp}_{\mathcal{K}}\left(k', \{k \sqcap_{\mathcal{K}} k' \mid (k, w) \in a \wedge k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}}\}\right)\} \end{aligned}$$

Note that this operator may lead to an infeasibly fine-grained segmentation, i.e. too many segments. And also with this operator ‘neighbouring’ segments, i.e. segments whose keys could be joined without changing the concretisation, may contain the same value abstraction, and thus dividing them would not have been necessary.

The strong update operator does not join the new value abstraction with any existing segment, but simply removes all overlapping parts of a and adds the new segment without changing it.

$$\begin{aligned} a \oplus (k', v') := & \left(a \setminus \{(k, w) \mid (k, w) \in a \text{ and } k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}}\}\right) \\ & \cup \{(m, w) \mid (k, w) \in a,\ k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}} \text{ and } m \in \mathit{decomp}_{\mathcal{K}}(k, k')\} \\ & \cup \{(k', v') \mid k' \neq \bot_{\mathcal{K}} \text{ and } v' \neq \bot_{\mathcal{V}}\} \end{aligned}$$

We also introduce a big version of this operator $\bigoplus$ which adds multiple abstract keys with the same abstract value by using $\oplus$:

$$a \bigoplus \{(k'_1, v'_1), (k'_2, v'_1), \dots, (k'_n, v'_1)\} := (((a \oplus (k'_1, v'_1)) \oplus (k'_2, v'_1)) \cdots) \oplus (k'_n, v'_1)$$

We allow the key abstractions to overlap, but since the added segments all have the same abstract value, the order in which we add them may only change the segmentation of the abstraction but not its concretisation.

3.2 The Domain

The overall domain is parametric in a key domain $\mathcal{K}$, a value domain $\mathcal{V}$ and a scalar domain $\mathcal{A}$. The scalar domain ranges over all scalar variables. The key domain and value domain (possibly) range over the scalar variables and an artificial variable v_k and v_v respectively, which represent the actual key or value abstraction for a segment. This representation is used to allow relational domains to express relationships between keys or values of a segment and scalar variables. Also this way normal scalar domains can be used without big modification. We require that the scalar abstractions in the segments always conform with the scalar domain abstraction, which means it must be equal to the scalar abstraction or an overapproximation of it.
Note that to be able to analyse lists the key domain should at least support an abstraction of integers.

For each of these domains Fulara’s domain expects the user to provide three functions to manipulate their variable environment. These are:

- A variable elimination function $remove(v, a)$, which removes a variable completely from the abstract state a,
- A variable introduction function $add(v, a)$, which adds a variable to the environment,
- And a 'forget' function $forget(v, a)$, which forgets the value of a variable (i.e. sets it to top).

Additionally, conversion functions are required in both directions between the scalar and the key or value domain. The functions are named $\kappa_{\mathcal{A}\to\mathcal{K}}, \kappa_{\mathcal{K}\to\mathcal{A}}$ and $\kappa_{\mathcal{A}\to\mathcal{V}}, \kappa_{\mathcal{V}\to\mathcal{A}}$ respectively.

The domain elements as described by Fulara consist of these three parts:

1. *Scalar abstraction*: An abstraction of scalar variables in some domain $\mathcal{A}$
2. *Content abstraction*: An abstraction of map data structure contents, mapping each map variable to a Segment Lattice
3. *Initialization abstraction*: An over-approximation of uninitialized map data structure elements, mapping each map variable to a Segment Lattice with boolean values as value domain

Formally the elements of the abstract domain are:

$$(a, c, i) \in \mathcal{A} \times (\mathbb{V}ar_{map} \to \mathcal{S}\,(\mathcal{K}, \mathcal{V})) \times (\mathbb{V}ar_{map} \to \mathcal{S}\,(\mathcal{K}, Bool))$$

The initialization information is used to get a more precise abstraction of initialized and uninitialized segments by combining an overapproximation of initialized segments in $\mathcal{S}\,(\mathcal{K}, \mathcal{V})$ with an overapproximation of uninitialized segments in $\mathcal{S}\,(\mathcal{K}, Bool)$ as follows:

- Definitely uninitialized concrete segments: Not abstracted in $\mathcal{S}\,(\mathcal{K}, \mathcal{V})$ (i.e. the value is $\perp_{\mathcal{V}}$)
- Possibly (un)initialized concrete segments: Abstract value is '`True`' in $\mathcal{S}\,(\mathcal{K}, Bool)$ and the segment is abstracted in $\mathcal{S}\,(\mathcal{K}, \mathcal{V})$
- Definitely initialized concrete segments: Not abstracted in $\mathcal{S}\,(\mathcal{K}, Bool)$ (i.e. the abstract value is '`False`') and the segment is abstracted in $\mathcal{S}\,(\mathcal{K}, \mathcal{V})$

3.2.1 Membership Abstraction

We add a fourth element to the domain, which is used to abstract membership relations introduced by conditions like 'x,y `in` d_1`.items()`', 'x `in` d_1`.keys()`' or 'y `in` d_1`.values()`'. Note that we also call these expressions a condition if they occur in a '`for`'-loop header (or exit condition).

This abstraction is used in the content analysis to refine coupled key and value variables, introduced by a '`.items()`'-condition, in conjunction. By that we mean refining one of them if the other gets refined by a condition, so that they abstract all key-value pairs for which the condition is possibly true. For instance, the condition at line 15 in Example 2 refines '`k`' to be greater or equal to 3 and so '`v`' will be refined to abstract all values that have a key satisfying this condition (namely 1 and 2).
More importantly, the membership abstraction will be used in the usage analysis to directly mark the corresponding abstract segments as used if a variable with a membership relation is used. As we will see this enables us to be more precise, since

```
 1 i: int = int(input())
 2 d: Dict[int, int] = {3: 2}
 3 v1: int = 0
 4 d[4] = 1
 5 d[v1] = 7
 6 d[1] = v1        # = 0
 7 v2: int = d[3]   # = 2
 8 d[2] = 3 + v2    # = 5
 9 # d = {0:7, 1:0, 2:5, 3:2, 4:1}
10 if i >= 2:
11     if i in d.keys():
12         v2 = d[i]
13 
14     for k, v in d.items():
15         if k >= 3:
16             v1 = v
17             d[k-2] = v1
```

Example 2: Example for Content Analysis

we can avoid transferring the usage information to the map data structure only at membership conditions, where we can only see if the scalar variables of the condition are used or not, and hence can only distinguish between the whole data structure being used or not.

We represent membership relations by a set of 3-tuples each containing a map variable as the target of the membership relation and two scalar variables whose value is contained in the keys or values of the map data structure respectively. To also cover conditions like 'x `in` d_1`.keys()`' and 'y `in` d_1`.values()`' we allow one of the scalar elements to be `None` (empty). In the example above line 14 would be represented by the tuple `(d,k,v)` and the condition in line 11 would be represented by `(d,i,None)`.

The lattice $\langle \mathcal{M}, \sqsubseteq_{in}, \sqcup_{in}, \sqcap_{in}, \bot_{in}, \top_{in}, \triangledown_{in} \rangle$ for this abstraction is formally defined as follows:

$$\mathcal{M} := \mathcal{P}\left(\mathbb{V}ar_{dict} \times \mathbb{V}ar_{scalar} \cup \{\texttt{None}\} \times \mathbb{V}ar_{scalar} \cup \{\texttt{None}\}\right),$$

$$\sqsubseteq_{in} := \supseteq, \qquad \sqcup_{in} := \cap, \qquad \sqcap_{in} := \cup, \qquad \bot_{in} := \mathcal{M}, \qquad \top_{in} := \emptyset, \qquad \triangledown_{in} := \sqcup_{in}$$

As shown above, the widening operator is equal to the join, since the domain is finite (the variable sets are finite). Also since it only uses syntactic elements that occur in the analyzed code, the abstraction cannot differ in multiple loop iterations.

Abstract Transformers. Since assignments overwrite the variable on the left-hand side, we forget all relations of this variable. If the right-hand side is also a variable, we additionally transfer all relations of the left-hand side to this variable. This means, for each relation containing the overwritten variable, we add a new tuple where we replace the overwritten variable with the assigned variable:

$$[\![x = s]\!]_{\mathcal{M}}(m) := \begin{cases} forget_{\mathcal{M}}(x, m) \cup \{(dl, x, u) \mid (dl, y, u) \in m\} \\ \qquad\qquad \cup \{(dl, u, x) \mid (dl, u, y) \in m\} & \text{if } s = y \in \mathbb{V}ar_{scalar} \\ forget_{\mathcal{M}}(x, m) & \text{otherwise} \end{cases}$$

$$[\![dl = me]\!]_{\mathcal{M}}(m) := \begin{cases} forget_{\mathcal{M}}(dl, m) \cup \{(dl, t, u) \mid (dl, t, u) \in m\} & \text{if } me = dl_1 \in \mathbb{V}ar_{dict} \\ forget_{\mathcal{M}}(c, m) & \text{otherwise} \end{cases}$$

$$[\![y = dl\texttt{[}x\texttt{]}]\!]_{\mathcal{M}}(m) := forget_{\mathcal{M}}(y, m)$$

Here we use t and u to refer to tuple elements that are either a scalar variable or ‘None’.

The $forget_{\mathcal{M}}$ function has the same meaning as for the other elements of the overall domain and is defined as follows:

$$\begin{aligned} forget_{\mathcal{M}}(v, m) := &\{(dl, t, u) \mid (dl, t, u) \in m \text{ and } v \notin \{dl, t, u\}\} \\ &\cup \{(dl, \mathsf{None}, y) \mid (dl, v, y) \in m \text{ and } y \neq \mathsf{None}\} \\ &\cup \{(dl, x, \mathsf{None}) \mid (dl, x, v) \in m \text{ and } x \neq \mathsf{None}\} \end{aligned}$$

We define a FILTER function which adds a tuple corresponding to the used condition or removes the relation for the negated condition:

$$\begin{aligned}
&\text{FILTER}_{\mathcal{M}}[\![y \texttt{ in } l]\!](m) := m \cup \{(l, \mathsf{None}, y)\} \\
&\text{FILTER}_{\mathcal{M}}[\![x \texttt{ in } d_1\texttt{.keys()}]\!](m) := m \cup \{(d_1, x, \mathsf{None})\} \\
&\text{FILTER}_{\mathcal{M}}[\![y \texttt{ in } d_1\texttt{.values()}]\!](m) := m \cup \{(d_1, \mathsf{None}, \text{y})\} \\
&\text{FILTER}_{\mathcal{M}}[\![x, y \texttt{ in } d_1\texttt{.items()}]\!](m) := m \cup \{(d_1, x, y)\} \\
&\text{FILTER}_{\mathcal{M}}[\![y \texttt{ not in } l]\!](m) := m \setminus \{(l, \mathsf{None}, y) \mid (l, \mathsf{None}, y) \in m\} \\
&\text{FILTER}_{\mathcal{M}}[\![x \texttt{ not in } d_1\texttt{.keys()}]\!](m) := m \setminus \{(d_1, x, u) \mid (d_1, x, u) \in m\} \\
&\qquad\qquad\qquad\qquad \cup \{(d_1, \mathsf{None}, y) \mid (d_1, x, y) \in m\} \\
&\text{FILTER}_{\mathcal{M}}[\![y \texttt{ not in } d_1\texttt{.values()}]\!](m) := m \setminus \{(d_1, t, y) \mid (d_1, t, y) \in m\} \\
&\qquad\qquad\qquad\qquad \cup \{(d_1, x, \mathsf{None}) \mid (d_1, x, y) \in m\} \\
&\text{FILTER}_{\mathcal{M}}[\![x, y \texttt{ not in } d_1\texttt{.items()}]\!](m) := m \setminus \{(d_1, t, y) \mid (d_1, t, y) \in m\} \\
&\qquad\qquad\qquad\qquad \setminus \{(d_1, x, u) \mid (d_1, x, u) \in m\} \\
&\qquad\qquad\qquad\qquad \cup \{(d_1, \mathsf{None}, z) \mid (d_1, x, z) \in m \wedge z \neq y \wedge z \neq \mathsf{None}\} \\
&\qquad\qquad\qquad\qquad \cup \{(d_1, z, \mathsf{None}) \mid (d_1, z, y) \in m \wedge z \neq x \wedge z \neq \mathsf{None}\}
\end{aligned}$$

The abstract transformers for loop and branch membership conditions are defined by just applying the FILTER function on the corresponding condition, except for ‘`for`’-loop headers. For them we need to apply the $forget_{\mathcal{M}}$ function on the loop variables first, since ‘`for`’-loop headers overwrite their old value:

$$\begin{aligned}
&[\![\texttt{for } y \texttt{ in } l]\!]_{\mathcal{M}}(m) := \text{FILTER}_{\mathcal{M}}[\![y \texttt{ in } l]\!](forget_{\mathcal{M}}(y, m)) \\
&[\![\texttt{for } x \texttt{ in } d_1\texttt{.keys()}]\!]_{\mathcal{M}}(m) := \text{FILTER}_{\mathcal{M}}[\![x \texttt{ in } d_1\texttt{.keys()}]\!](forget_{\mathcal{M}}(x, m)) \\
&[\![\texttt{for } y \texttt{ in } d_1\texttt{.values()}]\!]_{\mathcal{M}}(m) := \text{FILTER}_{\mathcal{M}}[\![y \texttt{ in } d_1\texttt{.values()}]\!](forget_{\mathcal{M}}(y, m)) \\
&[\![\texttt{for } x, y \texttt{ in } d_1\texttt{.items()}]\!]_{\mathcal{M}}(m) := \text{FILTER}_{\mathcal{M}}[\![x, y \texttt{ in } d_1\texttt{.items()}]\!](forget_{\mathcal{M}}(y, forget_{\mathcal{M}}(x, m)))
\end{aligned}$$

The rest of the transformers leaves the membership abstraction unchanged. Therefore we do not explicitly define them here.

3.2.2 Complete Domain

With the addition of the membership abstraction our domain elements are:

$$(a, c, i, m) \in \mathcal{C} := \mathcal{A} \times (Var_{Dict} \to \mathcal{S}(\mathcal{K}, \mathcal{V})) \times (Var_{Dict} \to \mathcal{S}(\mathcal{K}, Bool)) \times \mathcal{M}$$

All *lattice operations* of the overall domain, except for the widening operation, are performed point-wise, i.e. on each element of the domain's tuple. Because the widening of the segment lattice is relatively imprecise, the widening is first only applied to the scalar abstraction (and the rest only gets joined) until it does not change any more. Only then the map abstractions will get widened if it is still needed.

Remark: There will be one subtle issue, when using the *join* of the segment lattice $\mathcal{S}$ here, if the key or value domain contain an abstraction of scalar variables, which is not mentioned by Fulara. The problem will only occur if one of the two map abstractions to join contains a non-overlapping segment. In this case the segment will not be joined with any segment from the other abstraction and consequently also the scalar abstraction in this segment will not be joined. This means it may not conform with the joined scalar state in $\mathcal{A}$. Therefore one needs to join the scalar abstraction of non-overlapping segments with the scalar state of the other argument of the join, but one cannot easily access just the abstraction of all scalar variables without the artificial key or value variable when using a general, possibly relational domain. One option is to transform the scalar state into a key or value state setting the abstraction of the artificial variable to bottom and then joining it with the elements of the non-overlapping segment using the join of $\mathcal{K}$ or $\mathcal{V}$. At least for our framework this does not work, because often an abstraction of multiple variables is said to be completely bottom if one of its variables is abstracted as bottom. And then a general join function would just result in the non-bottom element without actually joining the abstractions 'variable by variable'. Therefore we solve this by requiring two special transformation functions $update_{\mathcal{K}}(k, s)$ and $update_{\mathcal{V}}(v, s)$ that are specific to the used key, value and scalar domains. They set the scalar part of a key or value abstraction to conform with a given scalar state. Hence, we apply them to non-overlapping segment after the scalar state has been joined.

3.3 Abstract Transformers

Assignments

Scalar Assignments. Scalar assignments, i.e. assignments only containing scalar variables, can be delegated to the scalar, key and value domain's transfer functions. We need to update the key and value abstractions of every segment in the content abstraction and the key abstraction of every segment in the initialization abstraction to update their relations to scalar variables. After that the segments may overlap (due to imprecision in the domains' operators), therefore we need to normalize them again with $dNorm$.

$$[\![x = s]\!]_{\mathcal{C}}(a, c, i, m) := ([\![x = s]\!]_{\mathcal{A}}(a),\ c',\ i',\ [\![x = s]\!]_{\mathcal{M}}(m))$$

where:

$$c' := \lambda dl.\ dNorm\left(\{([\![x = s]\!]_{\mathcal{K}}(k), [\![x = s]\!]_{\mathcal{V}}(w)) \mid (k, w) \in c(dl)\}\right)$$
$$i' := \lambda dl.\ dNorm\left(\{([\![x = s]\!]_{\mathcal{K}}(k), \texttt{True}) \mid (k, \texttt{True}) \in i(dl)\}\right)$$

Map Data Structure Assignments. Assignments to map variables update their whole abstraction. To be more precise we transfer the current scalar state into the added top segments.

$$
\begin{aligned}
&[\![l = \texttt{[]}]\!]_{\mathcal{C}}(a, c, i, m) := (a,\ c\,[l \mapsto \emptyset]\,,\ i\,[l \mapsto \{(top_{key},\ \texttt{True})\}]\,,\ [\![l = \texttt{[]}]\!]_{\mathcal{M}}(m))\\
&[\![d_1 = \texttt{\{\}}]\!]_{\mathcal{C}}(a, c, i, m) := (a,\ c\,[d_1 \mapsto \emptyset]\,,\ i\,[d_1 \mapsto \{(top_{key},\ \texttt{True})\}]\,,\ [\![d_1 = \texttt{\{\}}]\!]_{\mathcal{M}}(m))\\
&[\![l = \texttt{listinput()}]\!]_{\mathcal{C}}(a, c, i, m) := (a,\ c\,[l \mapsto \{(top_{key},\ top_{value})\}]\,,\\
&\qquad\qquad\qquad i\,[l \mapsto \{(top_{key},\ \texttt{True})\}]\,,\ [\![l = \texttt{listinput()}]\!]_{\mathcal{M}}(m))\\
&[\![d_1 = \texttt{dictinput()}]\!]_{\mathcal{C}}(a, c, i, m) := (a,\ c\,[d_1 \mapsto \{(top_{key},\ top_{value})\}]\,,\\
&\qquad\qquad\qquad i\,[d_1 \mapsto \{(top_{key},\ \texttt{True})\}]\,,\ [\![d_1 = \texttt{dictinput()}]\!]_{\mathcal{M}}(m))
\end{aligned}
$$

where:

$$top_{key} := \kappa_{\mathcal{A}\to\mathcal{K}}\,(add_{\mathcal{A}}(v_k, a)) \quad \text{and} \quad top_{value} := \kappa_{\mathcal{A}\to\mathcal{V}}\,(add_{\mathcal{A}}(v_v, a))$$

$$[\![dl_1 = dl_2]\!]_{\mathcal{C}}(a, c, i, m) := (a,\ c\,[dl_1 \mapsto c(dl_2)]\,,\ i\,[dl_1 \mapsto c(dl_2)]\,,\ [\![dl_1 = dl_2]\!]_{\mathcal{M}}(m))$$

Assignments with Subscriptions. The subscript x of a subscription $dl\texttt{[}x\texttt{]}$ in a given scalar state a is evaluated by adding v_k to the scalar state and assigning x to it using the scalar transformer:

$$\mathrm{k}(x; a) := \kappa_{\mathcal{A}\to\mathcal{K}}\,([\![v_k = x]\!]_{\mathcal{A}}\,(add_{\mathcal{A}}(v_k, a)))$$

Similarly we can evaluate the value abstraction of a scalar variable for an assignment to a subscription:

$$\mathrm{v}(y; a) := \kappa_{\mathcal{A}\to\mathcal{V}}\,([\![v_v = y]\!]_{\mathcal{A}}\,(add_{\mathcal{A}}(v_v, a)))$$

The scalar abstraction of values read from a subscription $dl\texttt{[}x\texttt{]}$ in scalar state a and content state c is determined by joining all values whose key abstraction overlaps with $k(x; a)$:

$$\mathrm{v}(dl\texttt{[}x\texttt{]}; a, c) := \kappa_{\mathcal{V}\to\mathcal{A}}\left(\bigsqcup\nolimits_{\mathcal{V}} \{w \mid (k, w) \in c(dl) \text{ and } \mathrm{k}(x; a) \sqcap_{\mathcal{K}} k \neq \bot_{\mathcal{K}}\}\right)$$

With these evaluation functions we can define the transformers as follows:

$$[\![y = dl\texttt{[}x\texttt{]}]\!]_{\mathcal{C}}(a, c, i, m) := (a',\ forget_{\mathcal{S}}(y, c),\ forget_{\mathcal{S}}(y, i),\ [\![y = dl\texttt{[}x\texttt{]}]\!]_{\mathcal{M}}(m))$$

where:

$$a' := remove_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathrm{v}(dl\texttt{[}x\texttt{]}; a, c) \sqcap_{\mathcal{A}} add_{\mathcal{A}}(v_v, a)))$$

For writes to a subscription we distinguish between *weak* and *strong* updates. To be able to do that the key domain needs to have a predicate $IsSingleton_{\mathcal{K}}(k)$ to determine if a given abstraction abstracts a single concrete key (is a '*singleton*'). If the key abstraction of the subscript is not a singleton, we will not be able overwrite the old value abstraction in overlapping segments and instead will have to just join the new value abstraction, because there will be only one value overwritten in the concrete and our abstraction will not know which. Otherwise our key abstraction will abstract one concrete key. Hence, we can overwrite the value in our abstraction and also update the initialization abstraction, since we can be sure the value at this key got written/initialized.
We use the operators to add a segment that we defined before for the segment lattice. For the weak update we have two possibilities with different precisions. Whereas

Fulara's weak update just joins all segments that overlap with the segment to add by using the $\otimes$-operator, we propose a different option, which keeps non-overlapping parts unchanged and also keeps the existing overlapping partitions unjoined by using the $\odot$-operator. The weak update in general looks as follows with the appropriate operator plugged in for $\bigcirc$:

$$[\![dl\texttt{[}x\texttt{]} \;= y]\!]^{(weak)}_{\mathcal{C}}(a, c, i, m) := (a,\; c',\; i, m) \qquad \text{if } \neg IsSingleton_{\mathcal{K}}(\mathrm{k}(x; a))$$

where:

$$c' := c[dl \mapsto c(dl) \bigcirc (\mathrm{k}(x; a), \mathrm{v}(y; a))], \qquad \bigcirc \in \{\otimes, \odot\}$$

The strong update uses the $\oplus$-operator to keep non-overlapping parts and replace the rest with the new segment.

$$[\![dl\texttt{[}x\texttt{]} \;= y]\!]^{(strong)}_{\mathcal{C}}(a, c, i, m) := (a,\; c',\; i', m) \qquad \text{if } IsSingleton_{\mathcal{K}}(\mathrm{k}(x; a))$$

where:

$$c' := c[dl \mapsto c(dl) \oplus (\mathrm{k}(x; a), \mathrm{v}(y; a))]$$
$$i' := i[dl \mapsto i(dl) \oplus (\mathrm{k}(x; a), \texttt{False})]$$

Conditions

All conditions are evaluated in the scalar domain, but we want to reflect the changes in the scalar part of the map data structure abstractions. Hence, for each of the following abstract transformers we define new content and initialisation abstractions in terms of the corresponding updated scalar state a':

$$c'(a') := \lambda dl_1. \left\{\left(update_{\mathcal{K}}(k, a'),\; update_{\mathcal{V}}(w, a')\right) \mid (k, w) \in c(dl_1)\right\}$$
$$i'(a') := \lambda dl_1. \left\{\left(update_{\mathcal{K}}(k, a'),\; \texttt{True}\right) \mid (k, \texttt{True}) \in i(dl_1)\right\}$$

Conditions containing subscriptions can be modelled by assigning the subscription to a (temporary) scalar variable before the condition and assigning it back after the condition. Then we can simply replace the subscription in the condition by the (temporary) variable and evaluate it like a scalar condition.

Scalar Conditions. Scalar conditions are evaluated by the transformers of the scalar domain.

$$[\![\texttt{if}\; sb]\!]_{\mathcal{C}}(a, c, i, m) := (a',\; c'(a'),\; i'(a'),\; m)$$

where:

$$a' := [\![\texttt{if}\; sb]\!]_{\mathcal{A}}(a)$$

Membership Conditions. We add the handling of Python's dictionary and list membership conditions. For that we need to compute the abstraction of all keys and values of a map data structure *dl* in the content state *c*:

$$\mathrm{k}_{all}(dl; c) := \kappa_{\mathcal{K}\to\mathcal{A}}\left(\bigsqcup\nolimits_{\mathcal{K}} \{k \mid (k, w) \in c(dl)\}\right)$$
$$\mathrm{v}_{all}(dl; c) := \kappa_{\mathcal{V}\to\mathcal{A}}\left(\bigsqcup\nolimits_{\mathcal{V}} \{w \mid (k, w) \in c(dl)\}\right)$$

For '`if`'-statements we compute the meet of all keys and values joined together with the abstraction of the corresponding scalar variable(s) and update the map data structure segments accordingly:

$$
\begin{aligned}
&[\![\texttt{if}\ y\ \texttt{in}\ l]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{if}\ y\ \texttt{in}\ l]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a' := a \sqcap_{\mathcal{A}} \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathrm{v}_{all}(l;c)))\\
&[\![\texttt{if}\ x\ \texttt{in}\ d_1\texttt{.keys()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{if}\ x\ \texttt{in}\ d_1\texttt{.keys()}]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a' := a \sqcap_{\mathcal{A}} \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathrm{k}_{all}(d_1;c)))\\
&[\![\texttt{if}\ y\ \texttt{in}\ d_1\texttt{.values()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{if}\ y\ \texttt{in}\ d_1\texttt{.values()}]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a' := a \sqcap_{\mathcal{A}} \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathrm{v}_{all}(d_1;c)))\\
&[\![\texttt{if}\ x,y\ \texttt{in}\ d_1\texttt{.items()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{if}\ x,y\ \texttt{in}\ d_1\texttt{.items()}]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a' := a\ \sqcap_{\mathcal{A}} \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathrm{k}_{all}(d_1;c)))\\
&\qquad\qquad\ \sqcap_{\mathcal{A}} \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathrm{v}_{all}(d_1;c)))
\end{aligned}
$$

'**for**'-loop headers overwrite the loop variables, and thus there is no meet with the old scalar abstraction of them. We just perform a meet before the assignment to keep track of the relations of v_k or v_v, similar to the abstract transformer for reads from a subscript.

$$
\begin{aligned}
&[\![\texttt{for}\ y\ \texttt{in}\ l]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ y\ \texttt{in}\ l]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a' := \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_v,a) \sqcap_{\mathcal{A}} \mathrm{v}_{all}(l;c)))\\
&[\![\texttt{for}\ x\ \texttt{in}\ d_1\texttt{.keys()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ x\ \texttt{in}\ d_1\texttt{.keys()}]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a' := \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_k,a) \sqcap_{\mathcal{A}} \mathrm{k}_{all}(d_1;c)))\\
&[\![\texttt{for}\ y\ \texttt{in}\ d_1\texttt{.values()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ y\ \texttt{in}\ d_1\texttt{.values()}]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a' := \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_v,a) \sqcap_{\mathcal{A}} \mathrm{v}_{all}(d_1;c)))\\
&[\![\texttt{for}\ x,y\ \texttt{in}\ d_1\texttt{.items()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ x,y\ \texttt{in}\ d_1\texttt{.items()}]\!]_{\mathcal{M}}(m))\\
&\quad\text{where:}\\
&\qquad a'' := \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_k,a) \sqcap_{\mathcal{A}} \mathrm{k}_{all}(d_1;c)))\\
&\qquad a' := \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_v,a'') \sqcap_{\mathcal{A}} \mathrm{v}_{all}(d_1;c)))
\end{aligned}
$$

The negation of these conditions in an '**if**'-statement is abstracted almost the same way, but instead of using the abstraction of all keys or values from the content abstraction the abstraction of all possibly uninitialized keys is used:

$$\mathrm{k}_{notin}(d_1;i) := \kappa_{\mathcal{K}\to\mathcal{A}}\left(\bigsqcup\nolimits_{\mathcal{K}} \{k \mid (k,\texttt{True}) \in i(d_1)\}\right)$$

They are the only keys for which the '**not in**'-condition is possibly true, because all other keys must be initialized by our choice of abstraction. Since we just have this information for keys, we can only refine our abstraction by conditions on them. For values we just overapproximate by leaving the scalar abstraction unchanged.

$[\![\texttt{if}\ y\ \texttt{not in}\ l]\!]_{\mathcal{C}}(a,c,i,m) := (a,\ c,\ i, [\![\texttt{if}\ y\ \texttt{not in}\ l]\!]_{\mathcal{M}}(m))$

$[\![\texttt{if}\ x\ \texttt{not in}\ d_1\texttt{.keys()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{if}\ x\ \texttt{in}\ d_1\texttt{.keys()}]\!]_{\mathcal{M}}(m))$

where:

$$a' := a\ \sqcap_{\mathcal{A}}\ \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathrm{k}_{notin}(d_1; i)))$$

$[\![\texttt{if}\ y\ \texttt{not in}\ d_1\texttt{.values()}]\!]_{\mathcal{C}}(a,c,i,m) := (a,\ c,\ i, [\![\texttt{if}\ y\ \texttt{not in}\ d_1\texttt{.values()}]\!]_{\mathcal{M}}(m))$

$[\![\texttt{if}\ x,y\ \texttt{not in}\ d_1\texttt{.items()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{if}\ x,y\ \texttt{not in}\ d_1\texttt{.items()}]\!]_{\mathcal{M}}(m))$

where:

$$a' := a\ \sqcap_{\mathcal{A}}\ \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathrm{k}_{notin}(d_1; i)))$$

For the negated conditions in '**for**'-loops, i.e. as the exit condition of such a loop we need to be aware, that the loop variables in Python keep the last value they had in the loop, but since there is no specified order for iterating a dictionary, the loop variables can have any value from inside the loop. Additionally, if the loop body is never executed, i.e. if the map data structure does not contain any value, the loop variables preserve the value they had before the loop. This means, we can only refine the value abstraction of loop variables for such conditions if we can be sure that the loop is never executed. We can check that by looking at the initialisation abstraction of the map variable. If the initialisation abstraction is not top, i.e. does not contain only a single segment which maps all keys to 'True', we know that there are keys whose values are definitely initialised. This means the loop is definitely executed. In this case we can apply the same transformation as for the '**in**'-condition, i.e. making the loop variables abstract all values or keys of the map data structure. Therefore we can define the abstract transformers as follows:

$[\![\texttt{for}\ y\ \texttt{not in}\ l]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ y\ \texttt{not in}\ l]\!]_{\mathcal{M}}(m))$

where:

$$a' := \begin{cases} \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_v, a) \sqcap_{\mathcal{A}} \mathrm{v}_{all}(l; c))) & \text{if } i(l) \neq (\top_{\mathcal{K}}, \texttt{True}) \\ a & \text{otherwise} \end{cases}$$

$[\![\texttt{for}\ x\ \texttt{not in}\ d_1\texttt{.keys()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ x\ \texttt{not in}\ d_1\texttt{.keys()}]\!]_{\mathcal{M}}(m))$

where:

$$a' := \begin{cases} \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_k, a) \sqcap_{\mathcal{A}} \mathrm{k}_{all}(d_1; c))) & \text{if } i(d_1) \neq (\top_{\mathcal{K}}, \texttt{True}) \\ a & \text{otherwise} \end{cases}$$

$[\![\texttt{for}\ y\ \texttt{not in}\ d_1\texttt{.values()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ y\ \texttt{not in}\ d_1\texttt{.values()}]\!]_{\mathcal{M}}(m))$

where:

$$a' := \begin{cases} \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_v, a) \sqcap_{\mathcal{A}} \mathrm{v}_{all}(d_1; c))) & \text{if } i(d_1) \neq (\top_{\mathcal{K}}, \texttt{True}) \\ a & \text{otherwise} \end{cases}$$

$[\![\texttt{for}\ x,y\ \texttt{not in}\ d_1\texttt{.items()}]\!]_{\mathcal{C}}(a,c,i,m) := (a',\ c'(a'),\ i'(a'), [\![\texttt{for}\ x,y\ \texttt{not in}\ d_1\texttt{.items()}]\!]_{\mathcal{M}}(m))$

where:

$$a'' := \mathit{remove}_{\mathcal{A}}\,(v_k,\ [\![x = v_k]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_k, a) \sqcap_{\mathcal{A}} \mathrm{k}_{all}(d_1; c)))$$

$$a' := \begin{cases} \mathit{remove}_{\mathcal{A}}\,(v_v,\ [\![y = v_v]\!]_{\mathcal{A}}\,(\mathit{add}_{\mathcal{A}}(v_v, a'') \sqcap_{\mathcal{A}} \mathrm{v}_{all}(d_1; c))) & \text{if } i(d_1) \neq (\top_{\mathcal{K}}, \texttt{True}) \\ a & \text{otherwise} \end{cases}$$

Note that we treat lists in the same way as dictionaries, although they have a specified order of iterating over them.

4 Map Data Structure Usage Analyses

Using the segment lattice, we introduced in Section 3, we designed two usage analyses for map data structures that work with the strongly live and the more precise usage abstraction respectively. These analyses are again parametric, but only in a key domain $\mathcal{K}_u$ that is defined the same way as for the content domain. The key domain can be different from the one used for the forward analysis, but then we need a conversion function that converts the other key domain into the key domain used here. We call this function $\kappa_{\mathcal{K}\to\mathcal{K}_u}$.

Domain Elements. As for the content analysis we have scalar usage abstraction, which works like the already developed usage analyses, and a map data structure abstraction that operates with the segment lattice. This means we map variables and abstract segments to usage abstractions. Formally the elements can be written as:

$$(u_\sigma, u_\mu) \in (\mathbb{V}ar_{scalar} \to \mathrm{U}) \times (\mathbb{V}ar_{map} \to \mathcal{S}(\mathcal{K}_u, \mathrm{U}))$$

Here U denotes the usage abstractions of the chosen usage domain, i.e. either $\{Live, Dead\}$ or $\{U, B, W, N\}$. Note that segments mapping to *Dead* or N are not contained in the segment lattice, since they are the bottom element of the corresponding usage domains.

The lattice operators and elements are defined as for the content analysis (almost point-wise).

Abstract Transformers. We run the content analysis as a preliminary analysis to be able to evaluate the corresponding keys for a value usage and precisely cover map data structure usage inside loops using the membership abstraction.
Hence, the abstract transformers take the result of the content analysis as a parameter: $[\![\cdot]\!]^{(a,c,i,m)}(u_\sigma, u_\mu)$. The normal argument is the usage domain element below the current statement, whereas the content domain element is the state *above* (i.e. before) the current statement.
We define the abstract transformers for the strongly live abstraction and the usage abstraction in the next two sections. While looking at the transformers recall that usage analyses are performed backwards.

4.1 Strongly Live Abstraction

4.1.1 Assignments

Scalar Assignments. Every variable occurring on the right-hand side of the assignment becomes *Live* if the scalar variable on the left-hand side is *Live*.
Here we also need to check if a variable on the right-hand side is in the membership relation abstraction m in the value position. If that is the case, we directly mark the corresponding segment as *Live*.
To update the map abstraction we need to know the key abstraction that corresponds to the used values. If the membership relation was introduced by a '`.items()`'-condition, we directly have the corresponding key abstraction as the scalar abstraction in a of the key variable contained in the membership tuple. We just need to convert it to the key domain. If we do not have a key variable in the membership tuple, we need to determine all possible keys for the value abstraction of the value variable. We do this by using the $K(y; a, c)$ function, which gives us all keys whose

values overlap with the value abstraction of y in the content domain.

$$\mathrm{K}(y; a, c) := \{k \mid (k, w) \in c(dl) \text{ and } w \sqcap_{\mathcal{V}} \mathrm{v}(y; a) \neq \bot_{\mathcal{V}}\}$$

For each of these key abstractions we add a tuple to our abstraction of usage that marks the corresponding values as *Live*. We do not need to distinguish between weak and strong updates, because all updates to *Live* are updates to the top element of the liveness abstraction, and thus with a strong update we abstract exactly the same concrete key-value pairs as with a weak update (with partitioning).
Additionally, we have to update the relations of the old key abstraction with scalar variables to incorporate the assignment. Since all variables in this statement are scalar, we can just use the abstract transformer of the key domain. Note that this is not the same type of transformer as for the key domain of the content analysis, since it needs to perform the update for a backward analysis.

$$[\![x = s]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u'_\mu)$$

where:

$$u'_\sigma := \lambda y. \begin{cases} \mathit{Live} & \text{if } u_\sigma(x) = \mathit{Live} \text{ and } y \in \mathit{vars}(s) \\ \mathit{Dead} & \text{if } y = x \text{ and } x \notin \mathit{vars}(s) \\ u_\sigma(y) & \text{otherwise} \end{cases}$$

$$u'_\mu := \lambda dl. \begin{cases} \begin{aligned} &(\mathit{dNorm}\,(\{([\![x = s]\!]_{\mathcal{K}_u}(k), u) \mid (k, u) \in u_\mu(dl)\}) \\ &\quad \bigoplus \{(\kappa_{\mathcal{K} \to \mathcal{K}_u}(\mathrm{k}(z; a)), \mathit{Live}) \mid \exists y \in \mathit{vars}(s).\ (dl, z, y) \in m\}) \\ &\quad \bigoplus \{ (\kappa_{\mathcal{K} \to \mathcal{K}_u}(k), \mathit{Live}) \mid \\ &\qquad \exists y \in \mathit{vars}(s).\ \exists (dl, \mathsf{None}, y) \in m.\ k \in K(y; a, c)\} \end{aligned} & \text{if } u_\sigma(x) = \mathit{Live} \\ u_\mu(dl) & \text{otherwise} \end{cases}$$

Map Data Structure Assignments. If a map data structure gets overwritten, the whole dictionary maps to *Dead* and, since this is the bottom element, it is omitted in the segment lattice. If another map data structure gets assigned, the liveness abstraction is copied to this map variable occurring on the right-hand side. This means it is *Live* (or *Dead*) for exactly the same key abstractions.

$$[\![dl = me]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma, u'_\mu)$$

where:

$$u'_\mu := \lambda dl_1. \begin{cases} u_\mu(dl) & \text{if } dl_1 = me \\ \{\} & \text{if } dl_1 = dl \text{ and } me \neq dl_1 \\ u_\mu(dl_1) & \mathit{otherwise} \end{cases}$$

Assignments with Subscriptions. Subscripts x are evaluated in the content domain by using the $\mathrm{k}(x; a)$ function and then converting it to the key domain used here. Additionally, we use the strong update operator $\oplus$ of the segment lattice. Again we do not need to distinguish between weak and strong updates.
We update the relations in the map segments as for the content analysis imprecisely by just forgetting variable y in the old segments.

$[\![y = dl[x]]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u'_\mu)$

where:

$$u'_\sigma := \lambda z. \begin{cases} \mathit{Live} & \text{if } u_\sigma(y) = \mathit{Live} \text{ and } z = x \\ \mathit{Dead} & \text{if } z = y \text{ and } y \neq x \\ u_\sigma(z) & \text{otherwise} \end{cases}$$

$$u'_\mu := \lambda dl_1. \begin{cases} \mathit{forget}_{\mathcal{S}}(y, u_\mu(dl_1)) \oplus (\kappa_{\mathcal{K}\to\mathcal{K}_u}(\mathrm{k}(x;a)), \mathit{Live})] & \text{if } dl_1 = dl \text{ and } u_\sigma(y) = \mathit{Live} \\ \mathit{forget}_{\mathcal{S}}(y, u_\mu(dl_1)) & \text{otherwise} \end{cases}$$

For assignments to a subscription $dl[x]$ we need to determine if the map data structure is *Live* for some of the keys abstracted by the subscript x. We introduce an auxiliary predicate *isLive* for this purpose, which is defined as follows:

$$\mathit{isLive}(dl[x]; u_\mu, a) := \exists (k, \mathit{Live}) \in u_\mu(dl).\ \ k \sqcap_{\mathcal{K}_u} \kappa_{\mathcal{K}\to\mathcal{K}_u}(\mathrm{k}(x;a)) \neq \bot_{\mathcal{K}_u}$$

If *isLive* is true, we mark the scalar variable in the subscript and the one that gets assigned as *Live*. We can only set the dictionary segment that gets overwritten to *Dead* if we can perform a strong update, i.e. if the key abstraction is a singleton. Otherwise we could perform a weak update, but it would leave the liveness abstraction of the dictionary unchanged.

$[\![dl[x] = y]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u'_\mu)$

where:

$$u'_\sigma := \lambda z. \begin{cases} \mathit{Live} & \text{if } \mathit{isLive}(dl[x]; u_\mu, a) \text{ and } z = x \text{ or } z = y \\ u_\sigma(z) & \text{otherwise} \end{cases}$$

$$u'_\mu := \begin{cases} u_\mu[dl \mapsto u_\mu(dl) \oplus (\kappa_{\mathcal{K}\to\mathcal{K}_u}(k(x;a)), \mathit{Dead})] & \text{if } \mathcal{S}_{\mathcal{K}_u}(\kappa_{\mathcal{K}\to\mathcal{K}_u}(\mathrm{k}(x;a))) \\ u_\mu & \text{otherwise} \end{cases}$$

4.1.2 Conditions

By definition the strongly live abstraction just marks every data item (i.e. variables and subscriptions) as *Live* that occurs in a condition. For membership conditions we deviate from this definition. In general we do not set the map variable of the condition to *Live*, since we treat the usage of it in a more fine-grained way by the abstract transformer for scalar assignments, but for '`if`'-conditionals over map data structure values the whole map data structure is always used, since the '`not in`' branch uses information about all elements of the data structure. For '`for`'-loops we also mark the loop variables as *Dead*, because either they get overwritten by an execution of the loop or the loop is never executed and then nothing is used in this 'branch'.

Scalar Conditions.

$[\![\texttt{if}\ sb]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u_\mu)$

where:

$$u'_\sigma := \lambda x. \begin{cases} \mathit{Live} & \text{if } x \in \mathit{vars}(sb) \\ \mathit{Dead} & \text{otherwise} \end{cases}$$

Membership Conditions. As described before, for '`in`'-conditions we only mark the scalar variables as *Live*, but for their negation we also need to mark the whole map data structure as *Live* if we condition on membership in the values of the data structure.

$$\llbracket \texttt{if } y \texttt{ in } \texttt{l} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[y \mapsto \mathit{Live}], u_\mu)$$
$$\llbracket \texttt{if } x \texttt{ in } d_1\texttt{.keys()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[x \mapsto \mathit{Live}], u_\mu)$$
$$\llbracket \texttt{if } y \texttt{ in } d_1\texttt{.values()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[y \mapsto \mathit{Live}], u_\mu)$$
$$\llbracket \texttt{if } x, y \texttt{ in } d_1\texttt{.items()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[x \mapsto \mathit{Live}][y \mapsto \mathit{Live}], u_\mu)$$
$$\llbracket \texttt{if } y \texttt{ not in } \texttt{l} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[y \mapsto \mathit{Live}], u_\mu[l \mapsto \{(\top_{\mathcal{K}_u}, \mathit{Live})\}])$$
$$\llbracket \texttt{if } x \texttt{ not in } d_1\texttt{.keys()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[x \mapsto \mathit{Live}], u_\mu)$$
$$\llbracket \texttt{if } y \texttt{ not in } d_1\texttt{.values()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[y \mapsto \mathit{Live}], u_\mu[d_1 \mapsto \{(\top_{\mathcal{K}_u}, \mathit{Live})\}])$$
$$\llbracket \texttt{if } x, y \texttt{ in } d_1\texttt{.items()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[x \mapsto \mathit{Live}][y \mapsto \mathit{Live}], u_\mu[d_1 \mapsto \{(\top_{\mathcal{K}_u}, \mathit{Live})\}])$$

For '`for`'-loop conditions we do the opposite and mark the loop variables as *Dead*.

$$\llbracket \texttt{for } y \texttt{ in } \texttt{l} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[y \mapsto \mathit{Dead}], u_\mu)$$
$$\llbracket \texttt{for } x \texttt{ in } d_1\texttt{.keys()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[x \mapsto \mathit{Dead}], u_\mu)$$
$$\llbracket \texttt{for } y \texttt{ in } d_1\texttt{.values()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[y \mapsto \mathit{Dead}], u_\mu)$$
$$\llbracket \texttt{for } x, y \texttt{ in } d_1\texttt{.items()} \rrbracket_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma[x \mapsto \mathit{Dead}][y \mapsto \mathit{Dead}], u_\mu)$$

For the negation of these conditions, i.e. the exit conditions, we do not change anything, since their 'branch' is always executed eventually.

4.2 Usage abstraction

4.2.1 Assignments

Scalar Assignments. In the scalar domain scalar assignments are performed the same way as in [7, 5]. Every variable occurring on the right-hand side of the assignment becomes U if the scalar variable on the left-hand side was used in this nesting level (U) or below (B). Additionally, if the variable we are assigning to does not occur on the right-hand side and it was U or B, we mark it as overwritten (W). As for the strongly live abstraction we also update the usage of map data structures if a variable of the right-hand side of the assignment is contained in the values of this map variable according to the membership abstractions. Furthermore, we update the key domain relations in the same way as for the strongly live analysis.

$\llbracket x = s \rrbracket_{\mathcal{U}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u'_\mu)$

where:

$$u'_\sigma := \lambda y. \begin{cases} U & \text{if } u_\sigma(x) \in \{U, B\} \text{ and } y \in \mathit{vars}(s) \\ W & \text{if } u_\sigma(x) \in \{U, B\},\ y = x \text{ and } x \notin \mathit{vars}(s) \\ u_\sigma(y) & \text{otherwise} \end{cases}$$

$$u'_\mu := \lambda dl. \begin{cases} \begin{array}{l} (\mathit{dNorm}\,(\{(\llbracket x = s \rrbracket_{\mathcal{K}_u}(k), u) \mid (k, u) \in u_\mu(dl)\}) \\ \bigoplus \{(\kappa_{\mathcal{K} \to \mathcal{K}_u}\,(\mathrm{k}(z; a)), U) \mid \exists y \in \mathit{vars}(s).\ (dl, z, y) \in m\}) \\ \bigoplus \{\,(\kappa_{\mathcal{K} \to \mathcal{K}_u}\,(k), U) \mid \\ \quad\quad \exists y \in \mathit{vars}(s). \exists (dl, \mathsf{None}, y) \in m. k \in K(y; a, c)\} \end{array} & \text{if } u_\sigma(x) \in \{U, B\} \\ u_\mu(dl) & \text{otherwise} \end{cases}$$

Map Data Structure Assignments. If a map data structure gets overwritten, every segment containing a U or B now gets marked as W, similar to the scalar case. To simplify the definition we can just set the usage abstraction in every segment to W, since we do not have N, the bottom element, in the segments and W segments will not be changed by this. If another map data structure gets assigned, the abstraction of its map variable becomes used for all key abstractions that map to U or B in the left-hand side abstraction. Recall that for updates to the top element we do not need to distinguish between weak and strong updates. For the updates we use the operator $\bigoplus$. By using this operator all parts that do not overlap with a U/B-segment from the left-hand side keep their usage abstraction. This is also following the same principle as the scalar assignment.

$\llbracket dl = me \rrbracket_{\mathcal{U}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u_\sigma, u'_\mu)$

where:

$$u'_\mu := \lambda dl_1. \begin{cases} u_\mu(dl_1) \bigoplus \{(k, U) \mid (k, U) \in u_\mu(dl) \vee (k, B) \in u_\mu(dl)\} & \text{if } dl_1 = me \\ \{(k, W) \mid (k, u) \in u_\mu(dl)\} & \text{if } dl_1 = dl \text{ and } me \neq dl_1 \\ u_\mu(dl_1) & \mathit{otherwise} \end{cases}$$

Assignments with Subscriptions. As for the strongly live abstraction subscripts x are evaluated in the content domain by using the $\mathrm{k}(x; a)$ function and then converting it to the key domain used here. Also we use the strong update operator $\oplus$ again for updates to the top element. As always the right-hand side gets used if the left-hand side was used in this scope or below.

The relations of map data structure segments to y are again only updated imprecisely by using $forget$.

$$[\![y = dl[x]]\!]_{\mathcal{U}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u'_\mu)$$

where:

$$u'_\sigma := \lambda z. \begin{cases} U & \text{if } u_\sigma(y) \in \{U, B\} \text{ and } z = x \\ W & \text{if } u_\sigma(y) \in \{U, B\}, z = y \text{ and } y \neq x \\ u_\sigma(z) & \text{otherwise} \end{cases}$$

$$u'_\mu := \lambda dl_1. \begin{cases} forget_{\mathcal{S}}(y, u_\mu(dl_1)) \oplus (\kappa_{\mathcal{K}\to\mathcal{K}_u}(\mathrm{k}(x; a)), U)] & \text{if } dl_1 = dl \text{ and } u_\sigma(y) \in \{U, B\} \\ forget_{\mathcal{S}}(y, u_\mu(dl_1)) & \text{otherwise} \end{cases}$$

For assignments to a subscription $dl[x]$ we again need to determine whether the abstraction of x overlaps with any segment that is used in this scope (U) or below (B). We introduce a predicate for this domain too:

$$isUsed(dl[x]; u_\mu, a) := \exists (k, u) \in u_\mu(dl).\ \ u \in \{U, B\} \wedge k \sqcap_{\mathcal{K}_u} \kappa_{\mathcal{K}\to\mathcal{K}_u}(\mathrm{k}(x; a)) \neq \bot_{\mathcal{K}_u}$$

Since the element $dl[x]$ gets overwritten, we want to mark it as W. In contrast to the strongly live case, we cannot omit the weak update, since we are not updating to the bottom element of the usage abstraction. Hence, we mark the overlap of every segment overlapping with the abstraction of the subscript x with the join of its old usage abstraction and W. We do not want to set unused parts to W. This means, we cannot directly apply the normal strong or weak update operators, which would also update unused, i.e. not present parts. For the strong update we just check with the $isUsed$ predicate if the singleton key abstraction overlaps with a U- or B-segment. For the weak update we define a modified operator:

$$\begin{aligned} a \odot^{(\text{only overlap})} (k', v') := & \big(a \setminus \{(k, w) \mid (k, w) \in a \text{ and } k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}}\}\big) \\ & \cup \{(m, w) \mid (k, w) \in a,\ k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}} \text{ and } m \in decomp_{\mathcal{K}}(k, k \sqcap_{\mathcal{K}} k')\} \\ & \cup \{(k \sqcap_{\mathcal{K}} k',\ w \sqcup_{\mathcal{V}} v') \mid (k, w) \in a \text{ and } k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}}\} \\ = & \ a \odot (k', v') \\ & \setminus \{(k'_{rest}, v') \mid k'_{rest} \in Decomp_{\mathcal{K}}(k', \{k \sqcap_{\mathcal{K}} k' \mid (k, w) \in a \wedge k \sqcap_{\mathcal{K}} k' \neq \bot_{\mathcal{K}}\})\} \end{aligned}$$

$$[\![dl[x] = y]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u'_\mu)$$

where:

$$u'_\sigma := \lambda z. \begin{cases} U & \text{if } isUsed(dl[x]; u_\mu, a) \text{ and } z = x \text{ or } z = y \\ u_\sigma(z) & \text{otherwise} \end{cases}$$

$$u'_\mu := \begin{cases} u_\mu[dl \mapsto u_\mu(dl) \oplus (\kappa_{\mathcal{K}\to\mathcal{K}_u}(k(x; a)), W)] & \text{if } IsSingleton_{\mathcal{K}_u}(\kappa_{\mathcal{K}\to\mathcal{K}_u}(k(x; a))) \\ & \quad \text{and } isUsed(dl[x]; u_\mu, a) \\ u_\mu[dl \mapsto u_\mu(dl) \odot^{(\text{only overlap})} (\kappa_{\mathcal{K}\to\mathcal{K}_u}(k(x; a)), W)] & \text{if } \neg IsSingleton_{\mathcal{K}_u}(\kappa_{\mathcal{K}\to\mathcal{K}_u}(k(x; a))) \\ u_\mu & \text{otherwise} \end{cases}$$

4.2.2 Conditions

The usage abstraction is more precise in the handling of conditions than the strongly live abstraction. It only marks data items (i.e. variables and subscriptions) occurring in conditions as used if there is a data item that is used (U) or overwritten (W) in the corresponding branch. We introduce a predicate for this condition:

$$\text{hasEffect}(u_\sigma, u_\mu) = (\exists x.\ u_\sigma(x) \in \{U, W\}) \ \vee\ (\exists dl.\ \exists (k, u) \in u_\mu(dl).\ u \in \{U, W\})$$

Again we treat membership conditions specially. We do not mark the map variable as used (except for **not in**-conditionals) and for '**for**'-loop conditions we mark the loop variables as overwritten (W), because these conditions are not restrictions on their previous value, but more like a sequence of overwrites.

Scalar Conditions.

$[\![\texttt{if}\ sb]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u_\mu)$

where:

$$u'_\sigma := \lambda x.\begin{cases} U & \text{if } x \in \mathit{vars}(sb) \text{ and hasEffect}(u_\sigma, u_\mu) \\ u_\sigma(x) & \text{otherwise} \end{cases}$$

Membership Conditions. For '**if**'-statements we mark the scalar variables as U and for their negations also the map data structure.

$$[\![\texttt{if}\ y\ \texttt{in}\ \texttt{l}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[y \mapsto U], u_\mu) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

$$[\![\texttt{if}\ x\ \texttt{in}\ d_1\texttt{.keys()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[x \mapsto U], u_\mu) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

$$[\![\texttt{if}\ y\ \texttt{in}\ d_1\texttt{.values()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[y \mapsto U], u_\mu) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

$$[\![\texttt{if}\ x,y\ \texttt{in}\ d_1\texttt{.items()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[x \mapsto U][y \mapsto U], u_\mu) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

$$[\![\texttt{if}\ y\ \texttt{not in}\ \texttt{l}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[y \mapsto U], u_\mu[l \mapsto \{(\top_{\mathcal{K}_u}, U)\}]) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

$$[\![\texttt{if}\ x\ \texttt{not in}\ d_1\texttt{.keys()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[x \mapsto U], u_\mu) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

$$[\![\texttt{if}\ y\ \texttt{not in}\ d_1\texttt{.values()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[y \mapsto U], u_\mu[d_1 \mapsto \{(\top_{\mathcal{K}_u}, U)\}]) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

$$[\![\texttt{if}\ x,y\ \texttt{not in}\ d_1\texttt{.items()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases} (u_\sigma[x \mapsto U][y \mapsto U], & \\ u_\mu[d_1 \mapsto \{(\top_{\mathcal{K}_u}, U)\}]) & \text{if hasEffect}(u_\sigma, u_\mu) \\ (u_\sigma, u_\mu) & \text{otherwise} \end{cases}$$

As for the strongly live abstraction the transformers for the negations work exactly the same.

For '**for**'-loop conditions we mark the scalar variables as W if the scalar variables were used (or used below) similar to an assignment. Otherwise we leave them unchanged.

$$[\![\texttt{for}\ y\ \texttt{in}\ \texttt{l}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases}(u_\sigma[y \mapsto W], u_\mu) & \text{if } u_\sigma(y) \in \{U, B\}\\ (u_\sigma, u_\mu) & \text{otherwise}\end{cases}$$

$$[\![\texttt{for}\ x\ \texttt{in}\ d_1\texttt{.keys()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases}((u_\sigma[x \mapsto W], u_\mu), u_\mu) & \text{if } u_\sigma(x) \in \{U, B\}\\ (u_\sigma, u_\mu) & \text{otherwise}\end{cases}$$

$$[\![\texttt{for}\ y\ \texttt{in}\ d_1\texttt{.values()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := \begin{cases}(u_\sigma[y \mapsto W], u_\mu) & \text{if } u_\sigma(y) \in \{U, B\}\\ (u_\sigma, u_\mu) & \text{otherwise}\end{cases}$$

$$[\![\texttt{for}\ x, y\ \texttt{in}\ d_1\texttt{.items()}]\!]_{\mathcal{L}}^{(a,c,i,m)}(u_\sigma, u_\mu) := (u'_\sigma, u_\mu)$$

where:

$$u'_\sigma = \lambda z. \begin{cases}W & \text{if } z \in \{x, y\} \text{ and } u_\sigma(z) \in \{U, B\}\\ u_\sigma(z) & \text{otherwise}\end{cases}$$

For the negation of these conditions, i.e. the exit conditions, we do not change anything, since their 'branch' is always executed eventually.

4.2.3 Usage Stack Operations

As for the usage analysis described in Section 2.2.2 we maintain a stack of usage abstractions for each nesting level, which operates as described in [7]. The only part we need to adapt are the functions to adapt the usage abstraction when we enter or leave a nesting level.

As described before, the INC function is used to compute a new usage abstraction from the old usage abstraction when we enter an inner nesting level. Our definition of the function performs exactly the same transformation of abstract usage properties as for the previously existing analysis, i.e. from U to B and from W to N. We need to be careful to not add a segment containing N, since it is the bottom element of our domain, and thus the segment would violate the definition of the segment lattice.

$$\textsc{inc}_{\mathcal{U}}((u_\sigma, u_\mu)) := (u'_\sigma, u'_\mu)$$

where:

$$u'_\sigma := \lambda x. \begin{cases}B & \text{if } u_\sigma(x) = U\\ N & \text{if } u_\sigma(x) = W\\ u_\sigma(x) & \text{otherwise}\end{cases}$$

$$u'_\mu := \lambda dl.\ \{(k, B) \mid (k, U) \in u_\mu(dl) \text{ or } (k, B) \in u_\mu(dl)\}$$

The DEC function is performed in the opposite case, when we exit an inner scope, to update the abstraction of the outer scope, which is given as the second argument. As for the inc we perform in principle the same updates as in the scalar case by keeping used or overwritten data items from the inner scope. For the segment lattices the segmentation of the inner and the outer scope may differ. Therefore we perform the update by using the strong update operator. We use the strong version, because we want to overwrite the abstraction of the outer scope. The order of the strong updates to W and U does not matter, because their key abstractions do not overlap by definition, since they all come from the inner segment lattice.

$$\text{DEC}_{\mathcal{U}}\left((u_\sigma^{(inner)}, u_\mu^{(inner)}), (u_\sigma^{(outer)}, u_\mu^{(outer)})\right) := (u'_\sigma, u'_\mu)$$

where:

$$u'_\sigma := \lambda x. \begin{cases} u_\sigma^{(inner)}(x) & \text{if } u_\sigma^{(inner)}(x) \in \{U, W\} \\ u_\sigma^{(outer)}(x) & \text{otherwise} \end{cases}$$

$$u'_\mu := \lambda dl. \left(u_\mu^{(outer)}(dl) \bigoplus \left\{(k, U) \mid (k, U) \in u_\mu^{(inner)}(dl)\right\}\right) \bigoplus \left\{(k, W) \mid (k, W) \in u_\mu^{(inner)}(dl)\right\}$$

5 Implementation and Evaluation

5.1 Implementation

We implemented the analyses in Python using the framework for abstract interpretation that Lyra provides. This framework did not support dictionaries. Hence, we first extended this framework to support them, for example by adding the dictionary type and support for dictionary membership conditions.

The structure of the added classes for the forward analysis and their correspondence in this report can be found in Appendix A.

In contrast to the three analyses described here we support (multiple) subscriptions on the right-hand side of assignments and in conditions. We evaluate these subscriptions by reading their value to a temporary scalar variable by using the transformer for reads described here. After this translation we end up with an expression only containing scalar variables, such that we can then pass the whole expression to the abstract transformers of the scalar domain. To transfer the relations, introduced by the evaluation in the scalar domain, to the map data structure we assign the temporary variables back to the corresponding subscript using the transformers for writes described in this report. After that we can remove all temporary variables.

For all functions that we require from the key, value and scalar domains that are not standard functionalities of abstract domains, we included three abstract classes that implement the Adapter design pattern. These serve as wrappers around normal scalar domains for which one wants to instantiate our domain. Additionally, all functions that deal with two or more domains, like the *update* or conversion functions, need to be given as arguments to the constructor of the content or usage domains.
Hence, to instantiate our domains for different concrete key, value and scalar domains one needs to provide subclasses of the wrapper classes for the three domains and implement the functions that are required as arguments for the content and usage domains.

5.2 Evaluation

We evaluated the usage analysis on some examples that we found on the internet and ones we created on our own. Since our implementation is not yet very widely applicable, we needed to evaluate some of them by hand.
In particular we only have an instantiation with the interval domain for keys, values and scalar variables. There are string domains in Lyra, but often they are not precise enough for the examples or they do not contain complements for all elements and so

the decomposition of keys is often not possible. Also our implementation is limited in terms of supported language constructs, like nested dictionaries. Therefore we will sometimes transform the examples into equivalent versions.
For the representation of results we only print the abstract value for v_k and v_v, respectively, since the domains used are not relational and we do not want to pollute the output with irrelevant information. Also for simplicity we only include the top element of the usage stack.

We start by the evaluation of the Example 1 presented in the introduction, which is inspired by existing code. For its evaluation using our domain with the usage abstraction we are only interested in improvements by the segmentation of map data structures. Hence, we exclude 'BUG C' and the corresponding lines 2, 13 and 14 for the sake of simplicity, since 'BUG C' can be detected without a fine-grained map data structure abstraction as we have seen in Section 2.
Note that we do not have an implementation for Python's '`defaultdict`' yet, but the same functionality can be implemented by initializing the dictionary as a normal empty dictionary and then including a conditional to check if the key we want to access is not yet present in the dictionary. If this is the case, we set the corresponding value to zero:

```
        if a not in score_occurrences.keys():
            score_occurrences[a] = 0
        score_occurrences[a] += weight
```

Workaround for defaultdict

In the code below we included the results of the analysis at some interesting points as comments:

```
scores: Dict[int, int] = dictinput()

score_occurrences = {}
# a -> W, b -> N, k -> W, scores_gt_10 -> W, weight -> W,
  score_occurrences -> {([100, inf], W)}, scores -> {}
for a, b in scores.items():
    if a < 100:
        weight: int = 3
        # a -> N, b -> N, k -> N, scores_gt_10 -> N, weight
          -> N, score_occurrences -> {([100, inf], B)},
          scores -> {}
    else:
        weight: int = 1
        if a not in score_occurrences.keys():
            score_occurrences[a] = 0
        # a -> U, b -> N, k -> N, scores_gt_10 -> N, weight
          -> U, score_occurrences -> {([100, inf], U)},
          scores -> {}
        score_occurrences[a] += weight   # BUG A & BUG B
        # a -> N, b -> N, k -> N, scores_gt_10 -> N, weight
          -> N, score_occurrences -> {([100, inf], B)},
          scores -> {}

# a -> N, b -> N, k -> W, scores_gt_10 -> U, weight -> N,
  score_occurrences -> {([100, inf], U)}, scores -> {}
```

```
18 for k in score_occurrences.keys():
19     if k > 10:
20         # a -> N, b -> N, k -> U, scores_gt_10 -> U, weight
             -> N, score_occurrences -> {([100, inf], U)},
             scores -> {}
21         scores_gt_10 += score_occurrences[k]
22 print(scores_gt_10)
23 # a -> N, b -> N, k -> N, scores_gt_10 -> N, weight -> N,
     score_occurrences -> {}, scores -> {}
```

Example 3: Results for Motivating Example

In line 21 ‘`score_occurrences`’ gets marked as used for keys greater or equal to 100, since the forward analysis inferred that the dictionary may only contain values for these keys due to the indentation bug in line 14.
Also note that due to ‘BUG A’ ‘`scores`’ does not become used in line 14, since only the keys represented by ‘`a`’ are used. This key usage is reflected by the change in the usage abstraction of ‘`a`’.
Hence, we could detect both errors: ‘BUG A’ causes the values of ‘`scores`’ to be unused and because of ‘BUG B’ not all values of ‘`score_occurrences`’ are possibly initialized.
If we resolve ‘BUG A’, we could see that only the ‘upper part’ of ‘`scores`’ is used due to ‘BUG B’. Without ‘BUG B’ ‘`score_occurrences`’ would be completely used, but the values of ‘`scores`’ would still be unused.

A small, but common example is the following, which we took from the BRCA challenge[5], a project for gene analysis to improve cancer detection. It copies items from a dictionary ‘`items`’ at indices given by another input dictionary to an output dictionary:

```
col_indices: Dict = dictinput()
items: Dict = dictinput()

bx_ids: Dict = {}
for key in col_indices.keys():
    temp: int = col_indices[key]
    bx_ids[key]: int = items[temp]
print(bx_ids)
```

Example 4: BRCA Example 1

Here we do not need our segmentation of the key space, since we do not know anything about the dictionaries which are given as input. Our usage analysis will just report that both input dictionaries are completely used and that the output dictionary is completely overwritten. A more precise analysis could cover, where the keys for a subscription come from, and thus could for example report that only the values of ‘`items`’ indexed by an element of ‘`col_indices`’ are used, which may not be all.

A common example from the field of machine learning is training multiple models on data and then performing a cross validation over their results[6]:

[5] cf. `https://github.com/BRCAChallenge/brca-exchange/blob/master/pipeline/data_merging/variant_merging.py` (last accessed on August 28, 2018), lines 406-410

[6] cf. `https://github.com/HealthCatalyst/healthcareai-py/blob/master/healthcareai/advanced_supvervised_model_trainer.py` (last accessed on August 28, 2018), lines 142-169

```python
model_names: List[str] = ['KNN', 'Logistic Regression', 'Random Forest Classifier']
cross_val_scores_by_name: Dict[str, List[int]] = dictinput()

accuracies: Dict[str, int] = {}
for name in model_names:
    scores = cross_val_scores_by_name[name]
    mean_acc = np.mean(scores)
    accuracies[name] = mean_acc

best_model, best_score = max(accuracies.items(), key=lambda t: t[1])
print(best_model + ':' + best_score)
```

Example 5: Cross Validation

Although only the best model and score are output, all elements of '`accuracies`' are used in the '`max`'-function. There may still be an issue, because the model names are hard-coded. Using a string or symbolic domain with enough precision as key domain, we could infer that only the scores of these three models in '`model_names`' are used. This may not be wanted if the input dictionary contains more than these three models.

We could not test our analysis for relational domains, because right now there is no such domain implemented in Lyra. Hence, there may be parts that are not sound for relational domains and probably it is not very precise for them.

6 Conclusion and Future Work

In this thesis we presented a generic framework to analyse map data structures using abstract interpretation which is parametric in a scalar, a key and a value domain. This framework includes a forward analysis which is based on a domain by Fulara [3]. This analysis is able to analyse the contents of a map data structures by segmenting their key space into different abstractions and mapping them to an abstraction of all possible values for that key abstraction. Additionally, the forward analysis is able to precisely abstract which keys of a map data structure have an initialised value.
Our additions to this analysis are mainly the support for Python's membership conditions and loops over map data structure elements. Also we improved the precision of the analysis.
Furthermore, the framework includes two backward analyses for data usage with different precisions which are able to analyse the usage of map data structure values. Like the forward analysis, the usage analyses partition the key space of a map data structure to allow representations of partial usage. To be able to infer the keys for a value usage these analyses make use of the results of the forward analysis which is run as a preliminary step.

The analyses can be applied to any data structure containing key-value pairs. As in Python internally many structures are represented by dictionaries, our analysis could for example even be applied to analyse attributes of instances by abstracting the '`__dict__`' attribute.

This generality comes at the cost of loosing efficiency. Especially copying the scalar abstraction in all segments to support relational domains is not very efficient. Also if we had an ordering on the key abstractions, many algorithms, for example for the join or strong and weak update operators, could be more efficient, since they would not need to iterate over all segments to check for overlaps.

Future Work. There are many possibilities to improve or augment the presented analyses.

- We could get more precise results for the usage of data items by combining the overapproximating analysis with an underapproximating analysis. By that we can specify objects that are definitely used and reduce the fraction of objects that *may* be used. The underapproximating analysis could work with (simple) numerical abstraction to see if different abstract inputs cause different abstractions for the outputs, which would mean that the input definitely has an effect on the outputs. For example one could use the *even-odd* domain on a simple program that just adds one to the input. Then an even (odd) input would lead to an odd (even) output, which would mean that the input is definitely used. As opposed to the presented syntactic analysis this is an semantic approach.
- To make the (usage) analyses more usable for real-world applications it could be made inter-procedural.
- Since our analyses do not support aliasing, i.e. accesses to the same data structure via multiple variables, they could be extended by an alias analysis.
- Also to make the analyses wider applicable one could add support for more language constructs of Python, like the '`add`' function for lists or the '`get`' function with a default value for dictionaries. Most of these constructs could be built by the language subset we used here. A more complicated extension may be to analyse nested dictionaries, but this could possibly be done by adding the map data structure domains themselves to the value domain.
- There is probably still some work needed to precisely support relational domains. For example the calls to the $forget$ functions could possibly be eliminated in some cases.
- The performance of the implementation could definitely be improved. For example there may be less copying needed, when using lists instead of a set of tuples for the representation of the segment set, since, in contrast to tuple (and set) elements, list elements are mutable, and thus could be directly updated by the domain's functions.
- Lists could be abstracted more precisely by covering that their indices range from 0 to one less than their length and that they are allocated contiguously.
- A simple extension would be to cover the usage of the length of map data structures. This would improve the precision of our output for example for loops over map data structures, where the map elements themselves are not used. In that case the map is not completely unused, since the length of the data structure could still have an effect on the output, because it determines the number of iterations.
- One could make more use of the initialisation abstraction. For example only the possibly initialised segments may be marked as used, when the whole map data structure is used.

References

[1] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *POPL*. ACM Press, 1977, pp. 238–252.

[2] Jędrzej Fulara. "Abstract Analysis of Numerical and Container Variables". PhD thesis. University of Warsaw, 2012. URL: `http://depotuw.ceon.pl/handle/item/239`.

[3] Jędrzej Fulara. "Generic Abstraction of Dictionaries and Arrays". In: *Electronic Notes in Theoretical Computer Science* 287 (2012). Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2012, pp. 53 –64. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2012.09.006`.

[4] U. Möncke R. Giegerich. "Invariance of Approximative Semantics with Respect to Program Transformations". In: *GI — 11. Jahrestagung*. Springer Berlin Heidelberg, 1981, pp. 1–10. URL: `https://doi.org/10.1007/978-3-662-01089-1_1`.

[5] C. Urban and P. Müller. "An Abstract Interpretation Framework for Input Data Usage". In: *European Symposium on Programming (ESOP)*. LNCS. Springer-Verlag, 2018, pp. 683–710.

[6] Caterina Urban. "Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes)". PhD thesis. École Normale Supérieure, Paris, France, 2015. URL: `https://tel.archives-ouvertes.fr/tel-01176641`.

[7] S. Wehrli. "Static Program Analysis of Data Usage Properties". MA thesis. ETH Zurich, Zurich, Switzerland, 2017. URL: `https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Simon_Wehrli_MA_report.pdf`.

A UML Class Diagrams

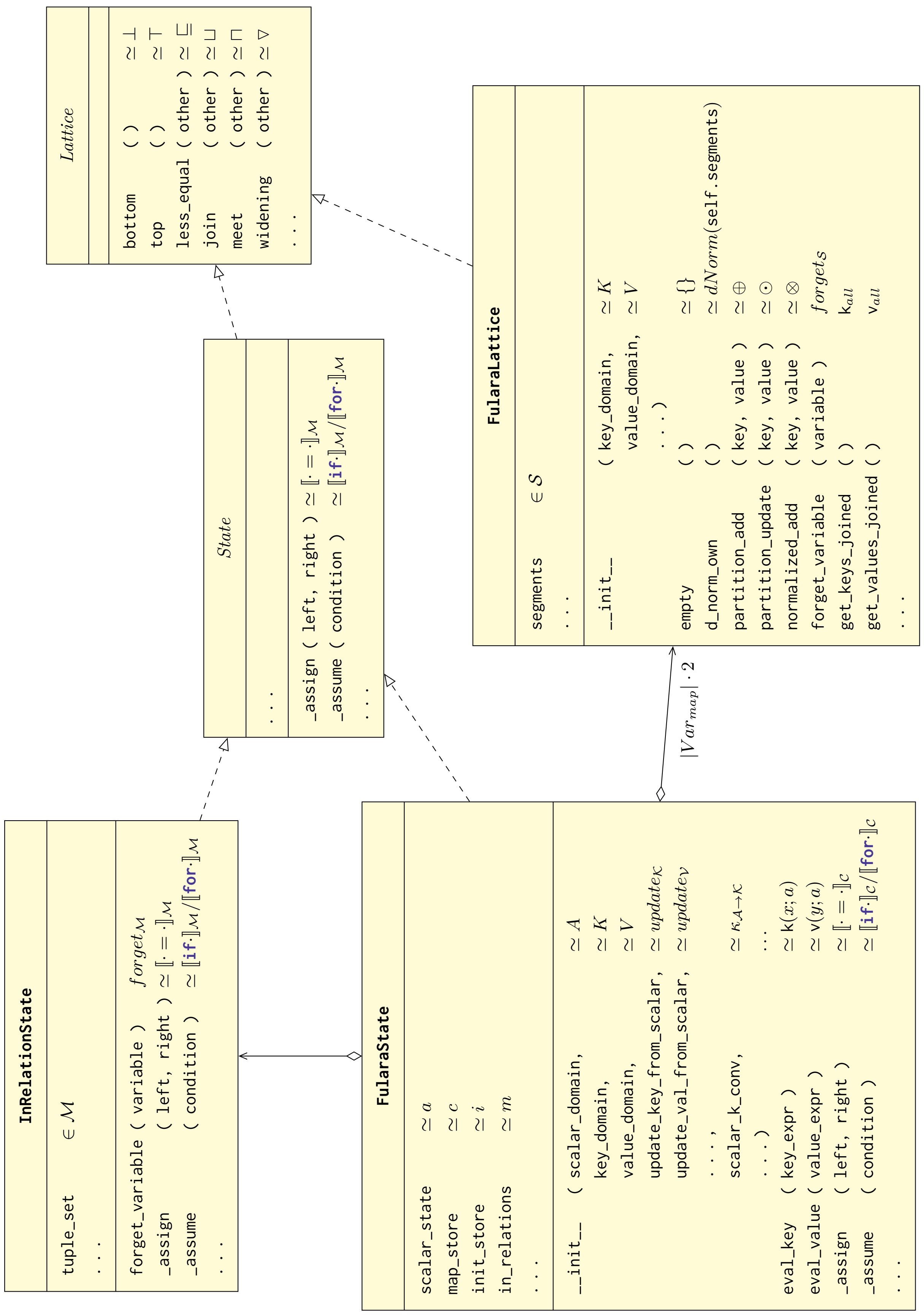
Lattice
bottom ( ) ≃ ⊥
top ( ) ≃ ⊤
less_equal (other) ≃ ⊑
join (other) ≃ ⊔
meet (other) ≃ ⊓
widening (other) ≃ ▽
. . .
State
. . .
_assign (left, right) ≃ ⟦· = ·⟧_M
_assume (condition) ≃ ⟦if·⟧_M/⟦for·⟧_M
. . .
FularaLattice
segments ∈ S
. . .
__init__ (key_domain, ≃ K
value_domain, ≃ V
. . .)
empty ( ) ≃ {}
d_norm_own ( ) ≃ dNorm(self.segments)
partition_add (key, value) ≃ ⊕
partition_update (key, value) ≃ ⊙
normalized_add (key, value) ≃ ⊗
forget_variable (variable) forget_S
get_keys_joined ( ) k_all
get_values_joined ( ) v_all
. . .
|Var_map| · 2
InRelationState
tuple_set ∈ M
. . .
forget_variable (variable) forget_M
_assign (left, right) ≃ ⟦· = ·⟧_M
_assume (condition) ≃ ⟦if·⟧_M/⟦for·⟧_M
. . .
FularaState
scalar_state ≃ a
map_store ≃ c
init_store ≃ i
in_relations ≃ m
. . .
__init__ (scalar_domain, ≃ A
key_domain, ≃ K
value_domain, ≃ V
update_key_from_scalar, ≃ update_K
update_val_from_scalar, ≃ update_V
. . . ,
scalar_k_conv, ≃ κ_{A→K}
. . .) . . .
eval_key (key_expr) ≃ k(x; a)
eval_value (value_expr) ≃ v(y; a)
_assign (left, right) ≃ ⟦· = ·⟧_C
_assume (condition) ≃ ⟦if·⟧_C/⟦for·⟧_C
. . .

Figure 2: Class Diagram for Content Analysis

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Usage Analysis of Data Stored in Map Data Structures

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Engel	Lowis Anton

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 28.08.2018	Engel

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.