Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Static Program Analysis of Data Usage Properties

Master Thesis

Simon Wehrli

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

Supervisors:
Dr. Caterina Urban
Prof. Dr. Peter Müller

2017

Abstract

We introduce a new property of program inputs that denotes if the input is *used* and develop a fully-automatic, static analysis within the framework of Abstract Interpretation to approximate this property for both single inputs and input lists of unknown length. The developed analysis extends the well-studied *truly-live variable analysis* to the stronger property of *usage*: It detects for input items, or more generally for any program variable, if they have any effect on the output of the program.

In Abstract Interpretation, an interpreter iterates over the code of a program without executing it on concrete inputs, and imitates parts of its behaviour with objects of an abstract universe, called abstract domain. Our analysis roots in an abstract *usage* domain, which holds information about which variables are used and how they can affect the usage property of other variables along the possible execution paths of the program. Combined with a stack-like structure, it can track the *flow of usage* through nested constructs like conditionals and loops. We can differentiate direct usage in the program output from implicit usage in conditions and transitive usage via assignments.

We then extend the usage domain to be able to track the usage property not only for variables, but also for list items. This means we can find out if the individual values of a list of inputs are used, even if the length of the list is statically unknown. We achieve this by integrating the usage domain into an abstract domain to analyze list contents. This domain is based on the idea of splitting the list into segments with dynamic bounds, each segment holding an abstraction of the values the segment is covering.

A prototype implementation of all presented domains and analysis was developed alongside this thesis. It demonstrates the potential usage scenarios in real-world programs typically written in data science applications.

Contents

1 Introduction

In the past years, the Python programming language and associated libraries became popular for data science applications due to the simple syntax and broad integration with common libraries for data analytics. For quick data inspection, the dynamic nature of Python seems very attractive, but introduces some challenges for analysis tools. Current tools mainly aim at detecting programming errors that lead to unhandled runtime exceptions. But programming errors that do not trigger exceptions at all can still have serious consequences. Erroneous treatment or exclusion of data items can lead to wrong results while showing no indication of the faulty derivation of those results.

Consider the case when we exclude data items unintentionally. Suppose we wrote a program to calculate the number of passing grades, i.e., grades greater or equal to 4, out of a list of grades provided as the input to the program. Code 1 shows a possible implementation. The structure of this program is typical for the sort of programs we tackle. In line 1 we read a list of inputs. The function listinput() reads one full line with an unknown number of input grades, which then are split and converted to a list of integers. Lines 5 to 11 count the passing grades. On the last line we print out the computed count.

```
1  list_grades = listinput()

   count = 0

5  i = 1  # Bug B: should be 'i = 0'
   while i < len(list_grades):
       g = list_grades[i]
       if g >= 4:
           g += 1  # Bug A: should be 'count += 1'
10
       i += 1

   print(count)
```

Code 1: A program with typical bugs we want to identify with our analysis.

The presented code contains two bugs: In line 5 we set the index i to the wrong starting value (Bug B) and in line 9 we increment the wrong variable (Bug A). Bug A let the whole while-loop be effectless on the result. On the other hand, Bug B alone (with Bug A fixed) let the first grade in the list be excluded from the counting.

The *live variable analysis* is used to separate live and dead variables in a program. A variable is dead in a line when its not live, and this is when it is assigned value is not read in all possible execution path after that line. In compiler design, this knowledge can be used for performance optimizations by omitting assignments to dead variables. An extension is the *truly-live variable analysis* [RHS95, GM81]:

Definition 1. Variable x is *truly-live* after a statement *stmt* (otherwise it is truly-dead/faint) if, and only if, there is a x-definition-free path from a successor of *stmt* to a statement *stmt*$'$ that reads the value of x and:

- *stmt*$'$ is a print statement, or
- *stmt*$'$ is a condition, or
- *stmt*$'$ is an assignemnt statement that assigns to a variable y that is truly-live after *stmt*$'$.

Whenever a variable is truly-live, it is also live, so truly-live is the stronger property.[1]

If we apply the truly-live variable analysis to our example, it will classify all variables as live. In fact, g is used in a condition on line 8, so Bug A will not be revealed. Detecting Bug B is out of the scope of this analysis, since it has no possibility to track liveness properties of list items individually and will denote the whole list as live as soon as one item is accessed on line 7.

Goals

In this thesis, we want to analyze a program for data usage properties, specifically, if an input data item is used in the calculations and has an (indirect) effect on the output. We want to find this with a static analysis of the program, so without presuming concrete input data that exposes a faulty behaviour. E.g., in Code 1 the analysis should detect that the first input grade has no effect on the printed result count, whatever the list of grades looks like in an actual execution. We target a sound approach that gives us mathematical guarantees of the specified properties about any syntactically correct program and input data. We will see that we choose our analysis to report unused inputs only if they are definitely unused in every actual execution.

We list the core goals of the thesis in detail and give an outline in which section we approach them:

- Define what is considered as an input data item to the analyzed program. We will see that we can handle inputs and variables in a similar fashion. (Section 2.2)
- Give a better intuition of the property "a data item is *used*" and formalize it. (Section 3.2)
- Design a static analysis within the framework of Abstract Interpretation [CC77] that approximates the solution to the question: Does a data item fulfill the *usage* property defined before? We build our analysis for a subset of Python (Section 2). The analysis has to be able to track the *flow of usage* through nested constructs like conditionals and loops. We should differentiate direct usage in the program output from implicit usage in conditions and transitive usage via assignments. First, we study this task on programs with scalar variables only (Section 5). We then extend the analysis to be able to track the usage property not only for variables, but also for list items. This means we can find out if the individual values of a list of inputs are used, even if the length of the list is statically unknown (Section 6 and 7).
- Implement the designed analysis (Section 8) and evaluate the analysis on serveral example programs (Section 8.3).

[1] The live variable property would only require x to be used in a reachable assignment *or* condition.

2 Language SimplePython

2.1 Assumptions

Some assumptions are made independently of the choice of a demonstration language. We assume that the programs analyzed

- are deterministic,
- are single-threaded,
- have a single pair of an initial and a final control point (before the initial and after the final statement, respectively),
- come type annotated[2].

2.2 Input/Output

A fully-automatic approach, so without the need for user input, is to assume that every line read from Python's standard input `sys.stdin` is a data item. The number of inputs is not statically known and we will not suppose that it is necessarily finite. This adds a real challenge. Related analysis, such as the mentioned truly-live variable analysis, assume that the number of items of interest is statically known (e.g., all variables in the program).

It is reasonable to assume that the input of a program consists of a known number of single input values or lists (of unknown length) of inputs. Even if the structure is more complex, the input reading procedure will often store the inputs in one of these forms. This implies that we can design our analysis to determine the usage property on the level of scalar variables and list items. The result of this broader analysis then can be easily matched to actual inputs and unused ones can be reported to the user. We still include the two input expressions `input()` and `listinput()` in the syntax defined in the next section, to emphasize the goal of our analysis to determine properties about input items.

2.3 Syntax

For the formal derivation and the implementation prototype, we use a subset of the Python language called *SimplePython*. The purpose is to have a simple language that is sufficient to show how the explained analysis work in practice. The subset also contains some constructs for presentational purposes not available in Python, but they have equivalents in Python (see the footnotes in the syntax definition below).

Python is strongly typed, so we will assume the types can be inferred by a pre-analysis or are explicitely provided by the user. For ease of presentation, SimplePython only supports integers and single-dimension lists of integers. We do not use the type annotations syntax in our examples, but use variable names that indicate the type of the associated objects (because there are only two possible). A variable also has a fixed type all over the program.

[2] Type annotations are available in Python ≥ 3.5, https://docs.python.org/3.5/library/typing.html

Constants:

$i \in \mathbb{Z}$	integer constants

Variables:

$x, y \in \mathbb{V}\text{ar}_{\text{int}}$	integer variables
$l \in \mathbb{V}\text{ar}_{\text{list}}$	list variables
$v \in \mathbb{V}\text{ar} = \mathbb{V}\text{ar}_{\text{int}} \cup \mathbb{V}\text{ar}_{\text{list}}$	general variables

Arithmetic expressions: (with and without list index)

$a, a_1, a_2 \in \mathbb{A}\text{expr}$ ::=	i	integer constants
|	x	integer variables
|	$l[a]$	list index access
|	input()	single integer input[3]
|	$-a$	unary operator
|	$a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$	binary operators[4]
$t \in \mathbb{A}\text{expr}^{-}$ ::=	$\mathbb{A}$expr without any list index accesses	

Boolean expressions:

$b, b_1, b_2 \in \mathbb{B}\text{expr}$ ::=	True	truth
|	False	falsity
|	**not** b	negation
|	b_1 **and** b_2	conjunction
|	b_1 **or** b_2	disjunction
|	$b_1 == b_2$	boolean comparison
|	$a_1 == a_2 \quad \mid \quad a_1 <= a_2 \quad \mid \quad a_1 < a_2$	arithmetic comparison

Expressions:

$e \in \mathbb{E}\text{xpr}$ ::=	$a \quad \mid \quad b$

List expressions:

$disp \in \mathbb{L}\text{istDisplaySeq}$ ::=	$a \quad \mid \quad disp, a$	list variable
$le \in \mathbb{L}\text{istExpr}$ ::=	l	list variable
|	$[disp]$	list display $[a_1, \ldots, a_n]$
|	[]	empty list display
|	listinput()	list input[5]

Statements:

$stmt \in \mathbb{S}\text{tmt}$	::=	**pass**	no effect, identity
	|	$x = a$	integer assignment
	|	$l = le$	list assignment
	|	$l[a_1] = a_2$	list index update
	|	$\text{print}(e)$	output
	|	**if** b : $stmt_1$ **else** $stmt_2$	conditional
	|	**while** b : $stmt$	iteration
	|	$stmt_1; stmt_2$ | $stmt_1$ <newline> $stmt_2$	sequence of statements

Program:

$prog \in \mathbb{P}\text{rog}$	::=	$stmt$	program

For the rest of this report, we use this naming scheme when we show formal programs and single lines of code, like expressions and statements in abstract transformers. Additionally we use $\psi \in \mathbb{V}\text{ar}_{\text{int}} \cup \mathbb{V}\text{ar}_{\text{list}} \cup \mathbb{I}\text{ndex}$, and $\omega \in \mathbb{E}\text{xpr} \cup \mathbb{L}\text{istDisplay}$. All presented code samples of this report conform to this syntax specification. Reformulating the indicated input expressions into the corresponding Python functions makes all of our code samples directly runnable by the Python interpreter.

We define a convenience operator on expressions to retrieve all the operands, since in the usage analysis we are only interested in the appearing operands, not in the operators:

$$
\begin{aligned}
\mathsf{opd}(i) &\stackrel{\text{def}}{=} i \\
\mathsf{opd}(x) &\stackrel{\text{def}}{=} x \\
\mathsf{opd}(l[a]) &\stackrel{\text{def}}{=} l[a] \\
\mathsf{opd}(-a) &\stackrel{\text{def}}{=} \mathsf{opd}(a) \\
\mathsf{opd}(a_1 \diamond a_2) &\stackrel{\text{def}}{=} \mathsf{opd}(a_1) \cup \mathsf{opd}(a_1) && \diamond \in \{+, -, *, /\} \\
\mathsf{opd}(a_1 \bowtie a_2) &\stackrel{\text{def}}{=} \mathsf{opd}(a_1) \cup \mathsf{opd}(a_1) && \bowtie \in \{==, <=, <\} \\
\mathsf{opd}(-b) &\stackrel{\text{def}}{=} \mathsf{opd}(b) \\
\mathsf{opd}(b_1 \square b_2) &\stackrel{\text{def}}{=} \mathsf{opd}(b_1) \cup \mathsf{opd}(b_1) && \square \in \{\ \textbf{and}\ ,\ \textbf{or}\ , ==\} \\
\mathsf{opd}(e) &\stackrel{\text{def}}{=} \varnothing && \text{for any other expression } e
\end{aligned}
$$

[3] The function input() does not return an integer in Python3, but this line can be replaced by int(input()) to get an integer input in Python3.

[4] Operator / denotes integer floor division here, which would be // in Python3.

[5] The function listinput() does not belong to the builtins of Python3, but can easily be implemented by list(map(int, input().split())).

3 Background

3.1 Abstract Interpretation

Our formal derivation and our implementation builds on the basis of *Abstract Interpretation*, a framework for approximating program behaviours [CC77]. Another detailed introduction can be found in the PhD thesis of a supervisor of this work [Urb15]. The key ingredients for every analysis built on Abstract Interpretation are described next:

Abstract Domains: One abstract domain (or several composed to one) allow representing the property of the program behaviour we are interested in. The representation is done by objects in some abstract universe, which holds partial information about the behaviour of an actual execution of the program. The concrete semantics describe this behaviour. We say *the concrete* when we are talking about actual execution behaviours. Note that, tracking all details of the concrete is not feasible for most programming languages. Therefore the decision what information to preserve in the abstract domains is crucial. A conretization function describes the meaning of the designed abstract objects in the concrete.

Lattices: Abstract domains should be based on a lattice structure and define a widening operator that is applied by the interpreter to guarantee termination of the analysis.

Abstract Semantics: Abstract semantics describe the effects of statements and expressions of the programming language on the abstract domain. The effects are described by *abstract transformers*, which are usually specific for the choosen abstract domain, and transform the abstract object, also called *state*. They imitate the effect of the concrete execution of the program on the abstract state without mapping the state into the concrete. We have to show that given abstract semantics are sound with respect to the concrete semantics.

Interpreter: The interpreter iterates over the program, applying the abstract transformers until a fixpoint is reached. It can be configured in several ways: some domains ask for a forward, others for a backward iteration, some for a combination. The interpreter may have to adjust the state additionally to how the abstract transformers change it, to reduce the number of iterations or prevent itself from running infinitely. This process is called *widening*.

There are two decisions to be made before designing the details of an abstract domain and its transformers, because abstract domains are usually sensitive to these decisions:

Type of Approximation: We can choose to under- or over-approximate the property of interest. It is the equivalent decision to prefer false-positives or false-negatives in the result set for which members we claim they satisfy the property of interest. An analysis that produces both false negatives and false positives is rarely desirable.

Forward/Backward Analysis: The interpreter can iterate through the program forwards or backwards. Some properties ask for a forward/backward analysis while others can be found both ways or sometimes a combination leads to the best results.

3.1.1 Octagon Domain

For our more sophisticated analysis, we use well-known numerical domains, which support us with numerical information about variables. Specifically, the Octagon Domain **Oct** [Min06] provides interval bounds, $b_l \leq x \leq b_u$, and relational constraints on every pair of variables, $x + y \leq c$, for variables $x, y \in \mathbb{V}\mathrm{ar}_{\mathrm{int}}, b_l, b_u, c \in \mathbb{Z}$. These constraints are useful to compare

expressions containing variables. We need such expressions to store dynamic information about which parts of lists are used.

When octagons are modified, they need to be *closed* to infer implicit constraints. The author of [Min06] presents a *tight closure* algorithm for the case of integer variables, but updating all constraints has a worst-case runtime complexity of $\mathcal{O}(n^4)$, where n is the number of variables. Therefore we use the improved *tight closure* algorithm from [BHZ08], which has the runtime complexity $\mathcal{O}(n^3)$. Their approach is basically a combination of the classical Floyd-Warshall shortest path algorithm with a tightening step[6] and a step to achieve strong coherence. During the algorithm, checks for different types of consistency are done and if one fails, the octagon is marked as infeasible (set to the bottom element).

Since the octagon domain only supports scalar variables, we use *weak assignments* for the list variables. This means we treat them as a single variable. Relational constraints with lists are not possible in the octagon domain. We do not need them because we use the octagons for the purpose of supporting our list content analysis. We only implemented it for integer variables and division is not supported yet.

3.1.2 Interval Domain

The interval domain [CC77] is a simple numerical domain that abstracts integer variables by an interval. We do not use the interval domain directly, since it is a strictly weaker abstraction compared to the octagon domain we use. However, we use the interval calculus as a fallback whenever the octagon can not answer queries about numerical properties because the query parameters can not be put into a form usable with octagonal constraints.

When we need the interval domain fallback, the strategy is always the same: We first build an interval abstract state from the information available in the octagon domain, specifically the worst-case bounds of a variable. Then we evaluate an expression in the interval calculus, using these bounds for variables and approximating other parts in the expressions by reasonable interval abstractions. E.g., an observed input() within an expression is abstracted by $[-\infty, \infty]$.

3.2 Usage Property

In this section, we formalize our intuition of the property, that an input is *used*. We first give an informal definition of the property we want to approximate:
Definition 2. Variable x is *used* after a statement *stmt* (otherwise *unused*), if and only if, there is a x-definition-free path from a successor of *stmt* to a statement $stmt'$ that reads the value of x and:

- $stmt'$ is a print statement, or
- $stmt'$ is a condition, that if satisfied, leads to a path that potentially has an effect on at least one output of the program, or
- $stmt'$ is an assignment statement that assigns to a variable y that is used after $stmt'$.

We suppose that each variable $x \in \mathbb{V}\mathrm{ar}$ has a (possibly infinite) domain D_i of valid values with an equivalence relation $=$. To ease explanation we assume that the fixed number of inputs and outputs are stored in specific variables $\mathbb{V}\mathrm{ar}_{\mathrm{In}} \subseteq \mathbb{V}\mathrm{ar}$ and $\mathbb{V}\mathrm{ar}_{\mathrm{Out}} \subseteq \mathbb{V}\mathrm{ar}$, respectively. As

[6] The pseudo-code in Figure 2 of [BHZ08] has an error in the tightening step: After applying the floor function, the result should again be doubled, i.e., the line should read $w[i, \bar{i}] := 2 * \mathsf{floor}(w[i, \bar{i}]/2)$;.

mentioned before, we can also determine the usage property for all variables, i.e., $\mathbb{V}\mathrm{ar}_{\mathrm{In}} = \mathbb{V}\mathrm{ar}$. The usage property is stored in a concrete environment $\rho \in \mathcal{E} = \mathbb{V}\mathrm{ar} \to \{U, N\}$, which holds for every variable if it is used (U) or unused (N).

Let Σ be the set of all program states. A program *state* $s \in \Sigma$ is a pair consisting of a program label and an environment that defines the values of the variables $\mathbb{V}\mathrm{ar}$ at the program control point designated by the label. We use the shorthand notation $s(x_i)$ to refer to the value of x_i in the environment belonging to state s. Let $\mathcal{I} \subseteq \Sigma$ be the set of *initial program states*, which are the states with a label designating the initial control point *before* the first statement of the program. Analogously, $\Omega \subseteq \Sigma$ is the set of *final program states* with a label designating the final control point *after* the final statement. A state t is called *reachable* from state s, denoted by the predicate $\mathsf{reach}(s, t)$, if there is a trace from s to t.

The variable x_i is considered used by a program if and only if there exist two initial program states $s, s' \in \mathcal{I}$ with equal values for all input variables except the input value for x_i and there are two final program states $t, t' \in \Omega$, *reachable* from s, s', respectively, with unequal values for at least one of the output variables. Formally,

$$\begin{aligned}\mathcal{E}(x_i) = U \;\equiv\; &\exists s, s' \in \mathcal{I}, t, t' \in \Omega : \mathsf{reach}(s, t) \wedge \mathsf{reach}(s', t') \\ &\wedge \forall x_j \in \mathbb{V}\mathrm{ar}_{\mathrm{In}} : (s(x_j) = s'(x_j) \leftrightarrow j \neq i) \wedge \exists o_k \in \mathbb{V}\mathrm{ar}_{\mathrm{Out}} : t(o_k) \neq t'(o_k).\end{aligned}$$

Similar properties have been stated and built analysis for in the context of secure information flow [BDR04].

Backwards Analysis As mentioned, the design of the abstract domains is sensitive to how the interpreter works. The property for data items we want to proof asks for a backward analysis, because the usage property propagates from the output statements/variables backwards to the inputs.

Over-Approximation We decided to target an analysis that produces an over-approximation of the set of used variables, i.e. we have no false-negatives, every variable denoted unused is definitely unused in the concrete.

4 Generic Functor Abstract Domains

We define reusable, composed domains in a generic way first. A functor[7] usually operates on at least one other domain $\mathbf{D}$, defined by the set of properties $\mathcal{D}$, induced lattice $L_D = \langle \mathcal{D}, \sqsubseteq_D, \sqcup_D, \sqcap_D, \bot_D, \top_D \rangle$ and a (incomplete[8]) set of abstract transformers. We do not introduce a domain and its elements separately all the time, but use different fonts with the same letter to distinguish the domain ($\mathbf{D}$), the set of abstract elements of the domain ($\mathcal{D}$) and use subscripts for operators ($\textsc{get}_D, \sqsubseteq_D$) and top/bottom elements ($\bot_D$). A functor can also take simple sets as (additional) parameters.

4.1 Abstract store domain

We often want to track certain values of entities (e.g., variables) in a program by an element of a lattice $L_{\mathcal{D}}$ of some abstraction $\mathbf{D}$, which motivates the definition of an *abstract store domain*[9]. The abstract domain functor $\mathbf{Store}(\mathcal{S}, \mathbf{D})$ takes a set $\mathcal{S}$ of entities to be tracked and serving as keys into the store. Second, it takes any abstract domain $\mathbf{D}$ and creates a new domain of abstract stores where each abstract store q is a mapping $\mathcal{S} \to \mathcal{D}$. An abstract store is also a complete lattice $L_{\mathcal{S}\to\mathcal{D}} = \langle \mathcal{S} \to \mathcal{D}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$, where the operators are defined by *pointwise lifting*:

$$
\begin{aligned}
q \dot{\sqsubseteq} q' &\stackrel{\text{def}}{=} \forall x \in \mathcal{S} : q(x) \sqsubseteq q(x)' \\
\dot{\bigsqcup} \mathcal{Q} &\stackrel{\text{def}}{=} \lambda x. \bigsqcup \{ q(x) | q \in \mathcal{Q} \} \\
\dot{\bigsqcap} \mathcal{Q} &\stackrel{\text{def}}{=} \lambda x. \bigsqcap \{ q(x) | q \in \mathcal{Q} \} \\
\dot{\bot} &\stackrel{\text{def}}{=} \lambda x. \bot \\
\dot{\top} &\stackrel{\text{def}}{=} \lambda x. \top
\end{aligned}
$$

We define a new formalism for updates on a store, extending the common store update $q[v \leftarrow d]$. We often want to update a store q to some value d at several keys $\mathcal{V} \subseteq \mathcal{S}$ at once, and we may want to do it based on some condition c (which may depend on the key). Therefore we introduce

$$
q\left[v \xleftarrow{c(v)} d \,\middle|\, v \in \mathcal{V} \right] \stackrel{\text{def}}{=} \dot{\bigsqcup} \left\{ q\left[v \leftarrow \begin{cases} d & \text{if } c(v) \\ q(v) & \text{otherwise} \end{cases} \right] \,\middle|\, v \in \mathcal{V} \right\}.
$$

When we need a store for different *types* of entities $\mathcal{S}_1, \mathcal{S}_2$, like different types of variables (integers/lists), we may also want them to map to different domains $\mathbf{D}_1, \mathbf{D}_2$. Therefore we define the union of stores as:

$$
\begin{aligned}
&\mathbf{Store}(\mathcal{S}_1, \mathbf{D}_2) \cup \mathbf{Store}(\mathcal{S}_1, \mathbf{D}_2) \stackrel{\text{def}}{=} \mathbf{Store}(\mathcal{S}_1 \cup \mathcal{S}_2, \mathbf{D}_1 \cup \mathbf{D}_2) = Q \\
&\text{s.t. for } q \in Q : (v \in \mathcal{S}_1 \Rightarrow q(v) \in \mathcal{D}_1) \ \wedge \ (v \in \mathcal{S}_2 \Rightarrow q(v) \in \mathcal{D}_1)
\end{aligned}
$$

The definition of the union $\mathbf{D}_1 \cup \mathbf{D}_2$ varies from case to case and has to provide a common partial order and a join/meet operator. By default we can assume the elements of $\mathcal{D}_1$ and $\mathcal{D}_2$ to be not comparable.

[7] The terminology of *functors* roots in the OCaml programming language and is explained in [CCL11].
[8] Some domains may only be useful to interpret a subset of a programs constructs.
[9] We call it *abstract store* as an opposite to *symbolic store*.

4.2 Abstract stack domain

We define a generic abstract domain functor **Stack**(Γ). Each instance can be described by a tuple $\langle \Gamma, \mathsf{push\_element}, \mathsf{pop\_element} \rangle$, where Γ is the set of valid elements on the stack (stack alphabet) and $\mathsf{push\_element} : \Gamma \to \Gamma$, $\mathsf{pop\_element} : \Gamma \times \Gamma \to \Gamma$ are funtions invoked when we push or pop an element, respectively. This two functions allow specific behaviour to be defined for a concrete stack instance.

Formally, a stack is defined as an (ordered) sequence $s \in \Gamma^*$ of stack elements, which form a complete lattice L_Γ. With $\epsilon \in \Gamma^*$, we denote the empty stack. A stack supports the following operations:

Push an element onto the stack:

$$\begin{aligned} \textsc{push} : \Gamma^k &\longrightarrow \Gamma^{k+1} \\ s &\longmapsto \langle \underbrace{s_1, \ldots, s_k}_{s}, \mathsf{push\_element}(s_k) \rangle \end{aligned}$$

Pop an element from the stack:

$$\begin{aligned} \textsc{pop} : \Gamma^{k+1} &\longrightarrow \Gamma^k \qquad (k \geq 1) \\ \langle s_1, \ldots, s_{k-1}, s_k, s_{k+1} \rangle &\longmapsto \langle s_1, \ldots, s_{k-1}, \mathsf{pop\_element}(s_k, s_{k+1}) \rangle \end{aligned}$$

The operators are defined by element-wise lifting of the operators defined for the domain of the elements on the stack. For two stacks $s, s' \in \Gamma^*$ with matching length, we define:

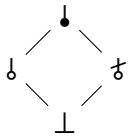
$$\begin{aligned} \langle s_1, \ldots, s_k \rangle \bar{\sqsubseteq} \langle s'_1, \ldots, s'_k \rangle &\stackrel{\text{def}}{=} \forall s_i . s_i \sqsubseteq s'_i \\ \langle s_1, \ldots, s_k \rangle \bar{\sqcup} \langle s'_1, \ldots, s'_k \rangle &\stackrel{\text{def}}{=} \langle s_1 \sqcup s'_1, \ldots, s_k \sqcup s'_k \rangle \\ \langle s_1, \ldots, s_k \rangle \bar{\sqcap} \langle s'_1, \ldots, s'_k \rangle &\stackrel{\text{def}}{=} \langle s_1 \sqcap s'_1, \ldots, s_k \sqcap s'_k \rangle \\ \bar{\bot} &\stackrel{\text{def}}{=} \langle \bot, \ldots, \bot \rangle \\ \bar{\top} &\stackrel{\text{def}}{=} \langle \top, \ldots, \top \rangle \end{aligned}$$

5 Usage Analysis for Integer Variables

5.1 Usage Abstract Domain

We introduce the *usage abstract domain* **U** which tracks for a scalar variable if it is used at a specific program point. Recall the intuitive Definition 2 of the usage property. One of the criterias to let a variable become used is that the variable appears in a condition that leads to a path that potentially has an effect on an output. Some paths only traverse straight-line code, others enter and exit conditionals and loops. To (implicitely) distinguish different types of paths, we need a notion of a *scope*. Eventhough our notion of scopes is inspired by *variable scopes*, they are not the same. Our scopes track the possibility of a branching condition in the control flow graph to have an effect on the output and are derived from the structure of the control flow graph. Since conditionals can be nested in SimplePython, scopes can also be nested, but all nested scopes must be fully contained in their parent scopes.

Let $L_{\mathcal{U}} = \langle \mathcal{U}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ be a complete lattice with the following Hasse-Diagram:



The elements $\mathcal{U} = \{⇃, ↓, ↯, \bot\}$ of this lattice have the following intuitive meaning:

$\top = ⇃$: Used in this scope or in a deeper nested scope.

$↓$: Used in an outer scope.

$↯$: Used in an outer scope and overriden in this scope.

$\bot$: Not used.

Lattice Operators The binary operators $\sqsubseteq, \sqcup, \sqcap$ are defined through the Hasse-Diagram above.

Additional Operators Descending into a nested scope (in a backwards fashion) requires to lower the abstract elements accordingly. Let $\textsc{descend} : \mathcal{U} \to \mathcal{U}$,

$\textsc{descend}(u)$	↓	↓	$\bot$	$\bot$
u	⇃	↓	↯	$\bot$

(1)

Exiting a nested scope (at its start because we do a backwards analysis) requires to combine the representation of the nested and the containing scope. Let $\textsc{combine} : \mathcal{U} \times \mathcal{U} \to \mathcal{U}$,

$\textsc{combine}(u, u')$	$\bot$	↓	↯	⇃	↓	↓	↯	⇃	↯	↓	↯	⇃	⇃	⇃	↯	⇃
u'	$\bot$	↓	↯	⇃	$\bot$	↓	↯	⇃	$\bot$	↓	↯	⇃	$\bot$	↓	↯	⇃
u	$\bot$	$\bot$	$\bot$	$\bot$	↓	↓	↓	↓	↯	↯	↯	↯	⇃	⇃	⇃	⇃

(2)

5.2 Usage Stores

We want to track all variables in a program with a value of the complete lattice $L_{\mathcal{U}}$, so we use abstract stores $q \in Q$, $q : \mathbb{V}\text{ar} \to \mathcal{U}$. This is an instantiation $\mathbf{Q} = \mathbf{Store}(\mathbb{V}\text{ar}, \mathbf{U})$ of the abstract store functor, hence the usage stores inherit the structure and they build also a complete lattice $L_Q = \langle \mathbb{V}\text{ar} \to \mathcal{U}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$.

Information Flow Operators

Implicit Flow Usage stores support the unary operator $\textsc{filter}_Q[\![e]\!] : Q \to Q$ used to update the abstract store on a branching condition in the control flow graph:

$$\textsc{filter}_Q[\![e]\!]q \stackrel{\text{def}}{=} \dot{\bigsqcup} \left\{ q\left[x \leftarrow \begin{cases} \downarrow & \text{if } \exists y \in \mathbb{V}\text{ar} : q(y) \in \{\downarrow, \not\downarrow\} \\ q(x) & \text{otherwise} \end{cases} \right] \middle| x \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar} \right\}$$

Explicit Flow The operator $\textsc{use}_Q[\![x = e]\!] : Q \to Q$ allows transitive usage of variables to be reflected in the abstract store:

$$\textsc{use}_Q[\![x = e]\!]q \stackrel{\text{def}}{=} \begin{cases} \dot{\bigsqcup} \{q[y \leftarrow \downarrow] | y \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}\} & \text{if } q(x) \in \{\downarrow, \not\downarrow\} \\ q & \text{otherwise} \end{cases}$$

Intermitted Flow The operator $\textsc{kill}_Q[\![x = e]\!] : Q \to Q$ permits the potential degradation of the abstract state of a variable at an overwriting assignment:

$$\textsc{kill}_Q[\![x = e]\!]q \stackrel{\text{def}}{=} q\left[x \xleftarrow{q(x) \in \{\downarrow, \not\downarrow\} \wedge x \notin \mathsf{opd}(e)} \not\downarrow \right]$$

Partial Abstract Semantics

$$\begin{aligned} [\![x = e]\!]q &\stackrel{\text{def}}{=} \left(\textsc{kill}_Q[\![x = e]\!] \circ \textsc{use}_Q[\![x = e]\!]\right) q \qquad \text{(order of application is crucial}^{10}\text{)} \\ [\![\text{print}(e)]\!]q &\stackrel{\text{def}}{=} \dot{\bigsqcup} \{q[x \leftarrow \downarrow] | x \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}\} \end{aligned} \tag{3}$$

5.3 Usage Stacks

We track all variables in a program with a usage property $u \in \mathcal{U}$ of the domain $\mathbf{U}$. The elements u implicitely track on what type of execution path they are rooted in. Therefore, they potentially change when the interpreter enters or exits a scope. To make the usage property of variables dependent on scopes, we use a stack instantiation $\mathbf{S} = \mathbf{Stack(Q)}$ with the configuration tuple $\langle Q, \textsc{descend}, \textsc{combine} \rangle$.

Lifting Operators We naturally lift the FILTER-operator on stores to stacks by simply modifying the top most element of a stack $s = \langle s_1, \ldots, s_k \rangle$:

$$\textsc{filter}_S[\![e]\!]s \stackrel{\text{def}}{=} \textsc{filter}_Q[\![e]\!]s_k$$

Abstract Transformers For what follows, we use $s, t \in S$ to denote abstract states.

[10] To avoid ambiguity: $(f \circ g)x \stackrel{\text{def}}{=} f(g(x))$

$$
\begin{aligned}
[\![\textbf{pass}]\!]s &\overset{\text{def}}{=} s \\
[\![x = e]\!]s &\overset{\text{def}}{=} \langle s_1, \ldots, s_{k-1}, [\![x = e]\!]s_k \rangle \\
[\![\text{print}(e)]\!]s &\overset{\text{def}}{=} \langle s_1, \ldots, s_{k-1}, [\![\text{print}(e)]\!]s_k \rangle \\
[\![\textbf{if } b : stmt_1 \textbf{ else } stmt_2]\!]s &\overset{\text{def}}{=} \text{POP}((\text{FILTER}_S[\![b]\!] \circ [\![stmt_1]\!])(\text{PUSH}(s))) \\
&\quad \dot{\sqcup}\ \text{POP}((\text{FILTER}_S[\![\textbf{ not } b]\!] \circ [\![stmt_2]\!])(\text{PUSH}(s))) \\
[\![\textbf{while } b : stmt]\!]s &\overset{\text{def}}{=} \mathsf{lfp}_s \left[\lambda t.\text{FILTER}_S[\![\textbf{ not } b]\!]t \right. \\
&\quad \left. \dot{\sqcup}\ \text{POP}((\text{FILTER}_S[\![b]\!] \circ [\![stmt]\!])(\text{PUSH}(t))) \right]
\end{aligned}
\tag{4}
$$

5.4 Concretization

The concretization γ_S of a stack depends on the concretization γ_Q of its element domain. In our analysis the purpose of the usage stacks is to remember usage properties accross scopes, so for the concrete only the top frame is relevant. For a stack s of lenght m we have:

$$
\gamma_S(s) \overset{\text{def}}{=} \gamma_Q(s_m) \tag{5}
$$

For the concretization of the usage store we reuse the usage property environment $\rho \in \mathcal{E} = \mathbb{V}\text{ar} \to \{U, N\}$ from Section 3.2. We define a function that returns the set of used variables in the concrete:

$$
\begin{aligned}
\text{USED} : \mathcal{E} &\longrightarrow \wp(\mathbb{V}\text{ar}) \\
\rho &\longmapsto \{o \in \mathbb{V}\text{ar} \mid \rho(o) = U\}
\end{aligned}
$$

The concretization of usage stores creates a environment with the same signature as a concrete environment:

$$
\begin{aligned}
\gamma_Q : Q &\longrightarrow \mathcal{E} \\
q &\longmapsto \lambda x. \begin{cases} U & \text{if } q(x) \in \{\downarrow, \downarrow\} \\ N & \text{otherwise} \end{cases}
\end{aligned}
$$

Theorem 1. The usage analysis is an over-approximation of the set of definitely used variables, i.e., the abstract state s and the concrete environment ρ at every program point satisfy:

$$
\text{USED}(\rho) \subseteq \text{USED}(\gamma_S(s))
$$

5.5 Example

Code 2 shows a rewrite of our program for counting passing grades (Code 1). It uses a fixed number of input grades read with input(). Bug A on line 8 leads to *count* not beeing updated upon the check that the math grade is passing. Bug B on line 11 leads to an update of *count* but, because of an erroneous check, involving the wrong variable *physics*. On line 6, our analysis correctly detects $count \to \downarrow, math \to \bot, physics \to \downarrow, history \to \bot$.

```
1  math = input()
   physics = input()
   history = input()

5  count = 0

   if math >= 4:
       math += 1  # Bug A: should be 'count += 1'
   if physics >= 4:
10     count += 1
   if physics >= 4:  # Bug B: should be 'history >= 4'
       count += 1

   print(count)
```

Code 2: Program with two bugs detectable with the **U** abstract domain. On line 6, the analysis outputs $count \to \text{⸸}, math \to \bot, physics \to \text{⸸}, history \to \bot$.

6 List Start Usage Analysis

In this section we explore the *list start usage domain*, denoted by **H**, to track the content of a list with respect to the usage property. More precisely, our domain will track how much from the start of a list is used, used outside of the current scope, overwritten or definitely not used, corresponding to the elements of $\mathcal{U}$. Since we want to track these properties *from the start* of the list, we only have to remember an upper limit until where the content of a list satisfies a property from $\mathcal{U}$. If we implicitely assume that the whole list is unused if no other property is stated, we only have to store the limits of the other properties $\downarrow, \underline{\downarrow}, \not\downarrow \in \mathcal{U}$. Thus, an element $h \in \mathsf{H}$, the set of all valid elements in the domain **H**, is a triple:

$$h = \underbrace{\langle h_{\underline{\downarrow}}, h_{\downarrow}, h_{\not\downarrow} \rangle}_{\text{indexed by } u \in \mathcal{U}} \in (\mathbb{N}_0 \cup \{\infty\})^3$$

The actual usage property at an index i of a list with abstraction h is the least upper bound of the properties *reaching* that index:

$$\textsc{get}_{\mathsf{H}}[h](i) = \textsc{get}_{\mathsf{H}}[\langle h_{\underline{\downarrow}}, h_{\downarrow}, h_{\not\downarrow} \rangle](i) \overset{\text{def}}{=} \bigsqcup (\{u \in \mathcal{U} | i < h_u\} \cup \{\bot\}) \tag{6}$$

> **Example 1.** Let a list l be abstracted by $h = \langle 0, 3, 5 \rangle$. This reads as the list items $l[0], l[1], l[2]$ are considered used ($\downarrow$) and $l[3], l[4]$ are considered overwritten ($\not\downarrow$). The rest of the list (of unknown length) is definitely unused. Figure 1 show a visualization of this example.

Whenever the limits $h_{\underline{\downarrow}}$ and $h_{\not\downarrow}$ disagree in h, i.e., when both are greater than 0, we say that h is not *coherent*. It is also not coherent when $h_{\underline{\downarrow}}$ and $h_{\not\downarrow}$ have no effect since they are smaller than $h_{\downarrow}$.

> **Example 2.** Both $h = \langle w, 3, 5 \rangle$ and $h' = \langle 0, 10, 5 \rangle$ are not coherent.

Definition 3. $h = \langle h_{\underline{\downarrow}}, h_{\downarrow}, h_{\not\downarrow} \rangle$ is *coherent* if $(h_{\underline{\downarrow}} = 0 \vee h_{\not\downarrow} = 0) \wedge h_{\downarrow} > \mathsf{max}(h_{\underline{\downarrow}}, h_{\not\downarrow})$. An alternative definition is that we require $\{u \in \mathcal{U} | i < h_u\}$ from (6) to contain at most one element.

Even though $\textsc{get}_{\mathsf{H}}$ resolves disagreeeing indices to $\downarrow$ (since $\underline{\downarrow}$ and $\not\downarrow$ are incomparable), we would like to have the elements always in a coherent state. Therefore we define the closure

$$\mathring{h} = \textsc{close}[h] = \textsc{close}[\langle h_{\underline{\downarrow}}, h_{\downarrow}, h_{\not\downarrow} \rangle] \overset{\text{def}}{=}$$
$$\left\langle \begin{cases} h_{\underline{\downarrow}} & \text{if } h_{\underline{\downarrow}} > h_{\not\downarrow}, h_{\downarrow} \\ 0 & \text{otherwise} \end{cases}, \mathsf{max}(h_{\downarrow}, \mathsf{min}(h_{\underline{\downarrow}}, h_{\not\downarrow})), \begin{cases} h_{\not\downarrow} & \text{if } h_{\not\downarrow} > h_{\underline{\downarrow}}, h_{\downarrow} \\ 0 & \text{otherwise} \end{cases} \right\rangle$$

Lemma 1. The closure $\textsc{close}[h]$ of any abstract element $h \in \mathsf{H}$ is *coherent*.

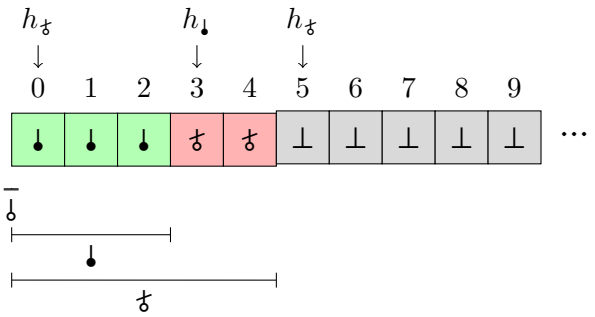


Figure 1: Visualization of the abstraction $h = \langle 0, 3, 5 \rangle$ in the list start usage domain **H**.

Analogously, we define an operator to represent an update of usage at a specific index with the most precise abstraction:

$$\begin{aligned}\textsc{set}_{\mathsf{H}}[h](i,u) =& \textsc{set}_{\mathsf{H}}[\langle h_{\text{🯅}}, h_{\text{🯆}}, h_{\text{🯄}}\rangle](i,u) \stackrel{\text{def}}{=} \textsc{close}\left[h'\right], \\ &\text{where } \forall \mu \in \mathcal{U} : h'_\mu = \begin{cases} \mathsf{max}(h_u, i) & \text{if } u = \mu \\ h_\mu & \text{otherwise} \end{cases}\end{aligned} \tag{7}$$

Lattice Operators We define the complete lattice $L_{\mathsf{H}} = \langle \mathsf{H}, \sqsubseteq_{\mathsf{H}}, \sqcup_{\mathsf{H}}, \sqcap_{\mathsf{H}}, \bot_{\mathsf{H}}, \top_{\mathsf{H}}\rangle$. The operators produce the same result as if we would apply them at each index individually and then find the most precise abstraction as a triple $h \in \mathsf{H}$:

$$\begin{aligned} h \sqsubseteq_{\mathsf{H}} h' &\stackrel{\text{def}}{=} \forall u \in \{\text{🯅}, \text{🯆}, \text{🯄}\} : \mathring{h}_u \leq \mathring{h}'_u \\ h \sqcup_{\mathsf{H}} h' &\stackrel{\text{def}}{=} \textsc{close}\left[\left\langle \mathsf{max}(\mathring{h}_{\text{🯅}}, \mathring{h}'_{\text{🯅}}), \mathsf{max}(\mathring{h}_{\text{🯆}}, \mathring{h}'_{\text{🯆}}), \mathsf{max}(\mathring{h}_{\text{🯄}}, \mathring{h}'_{\text{🯄}})\right\rangle\right] \\ h \sqcap_{\mathsf{H}} h' &\stackrel{\text{def}}{=} \textsc{close}\left[\left\langle \mathsf{min}(h_{\text{🯅}}, h'_{\text{🯅}}), \mathsf{min}(h_{\text{🯆}}, h'_{\text{🯆}}), \mathsf{min}(h_{\text{🯄}}, h'_{\text{🯄}})\right\rangle\right] \\ \bot_{\mathsf{H}} &\stackrel{\text{def}}{=} \langle 0, 0, 0\rangle \\ \top_{\mathsf{H}} &\stackrel{\text{def}}{=} \langle 0, \infty, 0\rangle \end{aligned}$$

The partial order and the meet operator require closed arguments for greatest possible precision and they all ensure that the result is again closed. The widening operator each of limits individually to infinity when their are growing and closes the result:

$$\begin{aligned} h \triangledown h' &\stackrel{\text{def}}{=} \textsc{close}\left[h^{\triangledown}\right] \\ &\text{where } \forall \mu \in \mathcal{U} : h^{\triangledown}_\mu = \begin{cases} \infty & \text{if } h_\mu < h'_\mu \\ h'_\mu & \text{otherwise} \end{cases} \end{aligned}$$

Lifting of Usage Abstract Domain Operators We lift the operators defined on $\mathcal{U}$ in **U** to the domain **H**. Recall the definition of DESCEND in (1). The result of DESCEND is bascially 🯅 if the argument was either 🯆 or 🯅, and $\bot$ otherwise. Thus we have to reset the limits of an abstraction h of domain **H** at both 🯆 and 🯄:

$$\textsc{descend}_{\mathsf{H}}(h) \stackrel{\text{def}}{=} \langle \mathsf{max}(h_{\text{🯅}}, h_{\text{🯆}}), 0, 0\rangle$$

Note that, for brevity, $\textsc{descend}_{\mathsf{H}}$ is given here *specifically* for the definition in (1). The lifting of the COMBINE operator to $\textsc{combine}_{\mathsf{H}}$ is fully generic without assuming anything about the definition of COMBINE[11]. Intuitively, the lifting finds all overlapping ranges and combines their properties with COMBINE and then maps this back to the most precise representation possible in **H**. To make the presentation easier, we assume $h_\bot = \infty$ is also defined for every $h \in \mathsf{H}$.

$$\begin{aligned} \textsc{combine}_{\mathsf{H}}(h, h') \stackrel{\text{def}}{=} \textsc{close}\Big[\Big\langle & \mathsf{max}\left\{ \mathsf{min}(h_u, h_{u'}) \middle| u, u' \in \mathcal{U} \wedge \textsc{combine}(u, u') = \text{🯅} \right\}, \\ & \mathsf{max}\left\{ \mathsf{min}(h_u, h_{u'}) \middle| u, u' \in \mathcal{U} \wedge \textsc{combine}(u, u') = \text{🯆} \right\}, \\ & \mathsf{max}\Big\{ \underbrace{\mathsf{min}(h_u, h_{u'})}_{\substack{\text{upper limit of} \\ \text{overlapping range}}} \Big| \underbrace{u, u' \in \mathcal{U} \wedge \textsc{combine}(u, u') = \text{🯄}}_{\text{all combinations resulting in 🯄}} \Big\} \Big\rangle \Big] \end{aligned}$$

[11] The implementation uses a generic way to realize this function, but is more efficient by not enumerating all the possible combinations $u, u' \in \mathcal{U}$ but using a scanline-algorithm.

6.1 Combining List Start Domain and Usage Domain

We track the scalar (integer) variables of the program with members of the usage domain $\mathbf{U}$ as in the integer-only programs from Section 5.2, and we track list variables with the list start domain $\mathbf{H}$. To combine these domains we use abstract stores $q \in Q = Q_{\text{int}} \cup Q_{\text{list}}$, where $Q_{\text{int}} = \mathbb{V}\text{ar}_{\text{int}} \to \mathcal{U}$ and $Q_{\text{list}} = \mathbb{V}\text{ar}_{\text{list}} \to \mathsf{H}$.

Information Flow Operators We extend usage stores to be able to filter expressions containing both scalar variable accessses and list index expressions. We define the condition when the operands in a filtered expression are considered used, i.e., when a store is *dirty*, as follows:

$$\text{DIRTY}(q) \stackrel{\text{def}}{=} \exists y \in \mathbb{V}\text{ar}_{\text{int}} : q(y) \in \{\downarrow, \not\downarrow\} \ \vee \ \exists l \in \mathbb{V}\text{ar}_{\text{list}} : q(l)_{\downarrow} > 0$$

Implicit Flow We account for implicit flow by checking if a store is dirty and updating the store accordingly:

$$\text{FILTER}_Q[\![e]\!]q \stackrel{\text{def}}{=} q\left[x \xleftarrow{\text{DIRTY}(q)} \downarrow \middle| x \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{int}}\right] \dot{\sqcup} \, q\left[l \xleftarrow{\text{DIRTY}(q)} \text{SET}_{\mathsf{H}}[q(l)](i;\downarrow) \middle| l[i] \in \mathsf{opd}(e) \cap \mathbb{I}\text{ndex}\right]$$

Explicit Flow We account for explicit flow by checking if left side of assignment is currently used:

$$\text{USE}_Q[\![x = e]\!]q \stackrel{\text{def}}{=} q\left[y \xleftarrow{q(x)\in\{\downarrow,\not\downarrow\}} \downarrow \middle| y \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{int}}\right] \dot{\sqcup} \, q\left[l \xleftarrow{q(x)\in\{\downarrow,\not\downarrow\}} \text{SET}_{\mathsf{H}}[q(l)](i;\downarrow) \middle| l[i] \in \mathsf{opd}(e) \cap \mathbb{I}\text{ndex}\right]$$

The transformer $\text{USE}_Q[\![l[i] = e]\!]q$ is similar except that the condition for updating the store is $\text{GET}_{\mathsf{H}}[q(l)](i) \in \{\downarrow, \not\downarrow\}$. When assigning to a list variable, we propagate the used property naturally to the right side of the assignment, whether it is another list variable or a list display expression:

$$\text{USE}_Q[\![l = l']\!]q \stackrel{\text{def}}{=} q\left[l' \leftarrow q(l)\left[p_i \xleftarrow{l\neq l' \wedge p_i\in\{\downarrow,\not\downarrow\}} \downarrow \middle| i \in \{0,\ldots,\mathsf{len}(l)\}\right]\right]$$

$$\text{USE}_Q[\![l = [t_0,\ldots,t_{m-1}]]\!]q \stackrel{\text{def}}{=} q\left[x \xleftarrow{\text{GET}_{\mathsf{H}}[q(l)](k)\in\{\downarrow,\not\downarrow\}} \downarrow \middle| x \in \mathsf{opd}(t_k), k \in \{0,\ldots,m-1\}\right]$$

Intermitted Flow The intermitted flow for an assignment to a scalar variable is as before:

$$\text{KILL}_Q[\![x = e]\!]q \stackrel{\text{def}}{=} q\left[x \xleftarrow{q(x)\in\{\downarrow,\not\downarrow\}\wedge x\notin\mathsf{opd}(e)} \not\downarrow\right]$$

We add the definitions for the assignment to list variables:

$$\text{KILL}_Q[\![l = l']\!]q \stackrel{\text{def}}{=} q\left[l \leftarrow q(l)\left[p_i \xleftarrow{l\neq l' \wedge p_i\in\{\downarrow,\not\downarrow\}} \not\downarrow \middle| i \in \{0,\ldots,\mathsf{len}(l)\}\right]\right]$$

$$\text{KILL}_Q[\![l = [t_0,\ldots,t_{m-1}]]\!]q \stackrel{\text{def}}{=} q\left[l \leftarrow q(l)\left[p_i \xleftarrow{p_i\in\{\downarrow,\not\downarrow\}} \not\downarrow \middle| i \in \{0,\ldots,\mathsf{len}(l)\}\right]\right]$$

To determine the intermitted flow for an assignment to a list index $l[i] = e$, we now take advantage of the simplification that i can only be a constant. This makes the check if the index i appears in the right side as easy as checking for an occurrence of $l[i]$[12]

$$\text{KILL}_Q[\![l[i] = e]\!]q \stackrel{\text{def}}{=} q\left[l \leftarrow \begin{cases} \text{SET}_{\mathsf{H}}[q(l)](i, \text{ɬ}) & \text{if } l[i] \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{list}} \\ q(l) & \text{otherwise} \end{cases}\right]$$

Finally we can combine this operators to adapt the partial abstract semantics given in (3) for int/list stores:

$$\begin{aligned} [\![\psi = \omega]\!]q &\stackrel{\text{def}}{=} \left(\text{KILL}_Q[\![\psi = \omega]\!] \circ \text{USE}_Q[\![\psi = \omega]\!]\right) q \\ [\![\text{print}(e)]\!]q &\stackrel{\text{def}}{=} q\,[x \leftarrow \text{ɬ} | x \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{int}}] \\ &\quad \dot{\sqcup}\, q\,[l \leftarrow \text{SET}_{\mathsf{H}}[q(l)](i; \text{ɬ}) | l[i] \in \mathsf{opd}(e) \cap \mathbb{I}\text{ndex}] \\ &\quad \dot{\sqcup}\, q\,[l \leftarrow \top_{\mathsf{H}} | l[i] \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{list}}] \end{aligned}$$

Abstract Transformers The abstract transformers stay the same as in (4) except that the assignment has support for lists:

$$[\![\psi = \omega]\!]s \stackrel{\text{def}}{=} \langle s_1, \ldots, s_{k-1}, [\![\psi = \omega]\!]s_k \rangle$$

6.2 Concretization

We focus on the conretization for the list start usage domain **H** here, the conretization of the combination with **U** is straight forward. For the concretization $\gamma_{Q,\mathsf{H}}$ of the part of the store that tracks list variables Q_{list}, we assume that each list element is identified by a distinct symbol $\sigma \in \mathbb{S}\text{ym}$ in the concrete. Then the conrete environment is a mapping from symbols the concrete properties used/unused, $\rho_{\text{list}} \in \mathcal{E}_{\text{sym}} = \mathbb{S}\text{ym} \to \{U, N\}$. The concretization is given by

$$\begin{aligned} &\gamma_{Q,\mathsf{H}} : Q \longrightarrow \mathcal{E}_{\text{sym}} \\ &\qquad q \longmapsto \lambda\sigma. \begin{cases} U & \text{if } \text{GET}_{\mathsf{H}}[q(l^\sigma)](i^\sigma) \in \{\text{ɬ}, \text{ɬ}\}, \\ N & \text{otherwise,} \end{cases} \end{aligned}$$

where l^σ is the list the symbol σ belongs to and i^σ is the corresponding index in the list.

We define the concretization of the stack s of length m in the same fashion as in (5):

$$\gamma_{S,H}(s) \stackrel{\text{def}}{=} \gamma_{Q,H}(s_m)$$

Theorem 2. The list start usage analysis includes an over-approximation of the set of list items definitely used, i.e., the abstract state s and the concrete environment ρ_{sym} at every program point satisfy:

$$\text{USED}(\rho_{\text{sym}}) \subseteq \text{USED}(\gamma_{S,H}(s))$$

[12] When we have access to an variable environment that can evaluate an expression a to an interval, we can also allow index expressions for $l[a]$. Refer to Section 7.3 for a more sophisticated domain accessing the variable environment.

6.3 Example

Code 3 shows a program to demonstrate the presented domain. The program takes two inputs, which are compiled with some constants to a list on line 4. On lines 8 to 11 we perform some random accesses to the list, using the accessed values in the printed sum. Since **H** can only track usage properties from the start of the list, we receive on line 5 that $list1$ is used up to index 4 ($h_{\downarrow}$ is exclusive):

$$list1 \rightarrow \langle 0, 5, 0 \rangle \tag{8}$$

On line 3 we get the correct result $x \rightarrow \downarrow, y \rightarrow \bot$. However, if we would remove line 9 (marker A), the analysis still outputs the same result even though x is no longer used, thus is imprecise.

```
1  x = int(input())
   y = int(input())

   list1 = [1, x, 2, 3, 5, 8, y]
5
   sum = 0
   # some random accesses to list1
   sum += list1[2]
   sum += list1[1]   # Marker A
10 sum += list1[4]
   sum += list1[0]
   print(sum)
```

Code 3: Program demonstrating the list start usage domain **H**.

7 List Content Usage Analysis

We first have a closer look at the ingredients, before combining them to segmentations.

7.1 Ingredients

7.1.1 Variable Environment Abstract Domain

In the context of a list content analysis the variables refering to lists are abstracted by a segmentation, so the variable environment must at least cover the abstractions for the set of scalar variables $\mathbb{V}\text{ar}_{\text{scalar}}$ in the program. This abstraction is known under the name *variable environment abstract domain*, denoted $\mathbf{V}$, and should comply with the interface of a $\mathbf{Store}(\mathbb{V}\text{ar}_{\text{scalar}}, \mathbf{D})$. Alternatively, we can use an extension that supports at least the retrival of some variable-specific properties, like the *Interval Domain* provides worst-case bounds for each scalar variable. We will build on a even more powerful, relational variable environment domain (*Octagon Domain*) and show some interesting modifications to operators on segmentations, as opposed to the proposal of [CCL11] to keep the segementation fully generic and not use any relational constraints on scalar variables. We still keep it as generic as possible and only require $\mathbf{V}$ to provide a partial order $\sqsubseteq_V$ that is capable of answering $v + c \sqsubseteq_V v' + c'$, where $v, v' \in \mathbb{V}\text{ar}$ and $c, c' \in \mathbb{Z}$.

7.1.2 Expression Abstract Domain

The expressions used for bounds have to be in some normal form. We will use the simple normal form $(\!|v + c|\!) \in E$[13], $v \in \mathbb{V}\text{ar}_{\text{int}} \cup \{v_0\}$ where $c \in \mathbb{Z}$ and v_0 is considered to be always equal to 0 (to represent a constant with $(\!|v_0 + c|\!)$).

Operators When we use expression in segmentation limits, we have to be able to decide about their equality. This can be done on a syntactic level only, by comparing the normal forms for equal variables and constants:

$$e = e' :\Leftrightarrow v = v' \wedge c = c' \qquad \text{(syntactic equality)}$$

But we can also use relational constraints from the variable environement abstract domain $\mathbf{V}$ to derive expression equality:

$$e \triangleq_E e' :\Leftrightarrow e \sqsubseteq_E e' \wedge e' \sqsubseteq_E e \qquad \text{(relational equality)}$$
$$\text{where } e \sqsubseteq_E e' :\Leftrightarrow v - v' \sqsubseteq_V c' - c \qquad \text{(partial order)}$$

Together with the least upper bound $\sqcup_E$ and greatest lower bound $\sqcap_E$ defined as usual, a bottom element $\bot_E$, representing unreachability, and a top element $\top_E$, representing expressions that can not be put in normal form, we can define the lattice $L_E = \langle E, \sqsubseteq_E, \sqcup_E, \sqcap_E, \bot_E, \top_E \rangle$.

7.1.3 Segment Limit Abstract Domain

Each segment limit in $\mathbf{L}(\mathbf{E_V})$ is a non-empty set of expressions from $\mathbf{E_V}$. All the bounds inside a limit set are supposed to be pairwise equals in the concrete. For limits l, l', we lift the equality

[13] The parenthesis are not shown in the examples, only when needed for clarity.

definitions from the expression domain:

$$l = l' :\Leftrightarrow \forall b \in l : b \in l' \wedge \forall b' \in l' : b' \in l \qquad \text{(syntactic equality } \triangleq \text{ set equality of } l, l')$$
$$l \sqsubseteq_L l' :\Leftrightarrow \exists b \in l, b' \in l' : b \sqsubseteq_E b' \qquad \text{(partial order)}$$
$$l \triangleq_L l' :\Leftrightarrow \exists b \in l, b' \in l' : b \triangleq_E b' \qquad \text{(relational equality)}$$

Eventhough we could add a top and bottom element to complete the lattice of limits, it is not necessary since we will remove limits whenever a bound turns into $\bot_E$ or $\top_E$.

7.1.4 Predicate Abstract Domain

We will use the simplest possible abstraction, where the concrete indices of the list covered by an abstract segment is abstracted by a single element, called predicate, from a domain $\mathbf{P}$. We assume the lattice $L_P = \langle P, \sqsubseteq_P, \sqcup_P, \sqcap_P, \bot_P, \top_P \rangle$ is given. One example for $\mathbf{P}$ is our usage property domain $\mathbf{U}$ from Section 5, which we will use as the predicate domain instantiation in Section 7.3. It is possible to use a more complex list element abstraction that relates to the list element index or even the segment index.

7.2 Generic List Segmentation Abstract Domain

We now combine the domains introduced to build the list segmentation abstract domain with the signature $\mathbf{Seg}(\mathbf{L}(\mathbf{E_V}), \mathbf{P}, \mathbf{V})$. A segmentation g has the form

$$\underbrace{\left\{b_1^0 \dots b_{m^0}^0\right\}}_{L_0} p_0[?]_0 \underbrace{\left\{b_1^1 \dots b_{m^1}^1\right\}}_{L_1} p_1[?]_1 \quad \cdots \quad p_{n-1}[?]_{n-1} \underbrace{\left\{b_1^n \dots b_{m^n}^n\right\}}_{L_n},$$

where $b_j^i \in E$, $i \in \{0, \dots, n\}, j \in \{0, \dots, m^j\}$ and $p_k \in P, [?]_k \in \{?, \_\}$, $k \in \{0, \dots, n-1\}$. We use $\mathsf{len}(g) = n$ to denote the length of the segmentation. Note that, the number of limits is $\mathsf{len}(g)+1$. We use the notation $g.p_i$ to retrieve the predicate at index i. We introduce a formlism for one or multiple updates of predicates at indices $\mathcal{I}$ to the new predicate P, conditioned on c (may depend on the old predicate at index i):

$$g\left[p_i \xleftarrow{c(p_i)} P \middle| i \in \mathcal{I}\right] \stackrel{\text{def}}{=} L_0 p_0'[?]_0 L_1 p_1'[?]_1 \quad \cdots \quad p_{n-1}'[?]_{n-1}$$
$$\text{where } p_i' = \begin{cases} P & \text{if } i \in \mathcal{I} \wedge c(p_i) \\ p_i & \text{otherwise} \end{cases}$$

This is similar to the formalism for stores in Section 4.1.

The interpretation of a segment is that the predicate p_i covers the segment range in the concrete. The upper bound is always exclusive while the lower bound is inclusive except when the segment is empty, i.e., when the limits coincide. The question mark $[?]_k$ after each predicate is either present (?) or absent (_) and denotes if the segment can possibly be empty or we know that at least one bound in the left limit is strictly smaller than a bound in the right limit:

$$[?]_k = \begin{cases} \_ & \text{if } \exists b \in L_k, b' \in L_{k+1} : b \sqsubseteq_E b' \\ ? & \text{otherwise} \end{cases}$$

In case two limits actually coincide in the concrete, the segment has zero length and the abstract predicate of that segment has no effect on the concrete. With the order $\_ \preccurlyeq ?$ and the least

upper bound $\curlyvee$ and greatest lower bound $\curlywedge$ defined as usual we get the lattice $L_? = \langle \{?, \_\}, \preccurlyeq, \curlyvee, \curlywedge, \_, ? \rangle$.

The consecutive, non-overlapping segments cover the whole list at any time, which means that the extremal limits are supposed to be at the limits of the concrete list. They are never removed and the invariant holds, that $0 \in L_0$ and $list\_len \in L_n$. This implies that there is always at least one limit in the segmentation (since the two extremal limits can be equal for an empty list).

7.2.1 Segmentation Unification

To be able to define a partial order and join, meet, widening, narrowing operators on segmentations we need a way to *unify* two segmentations g, g'. After unifying, they must have the same number and equal limits at each index, i.e., $\forall i \in \{0, \ldots, n\} : L_i = L'_i$, where at this point $n = \mathsf{len}(L_i) = \mathsf{len}(L'_i)$. The operators can then be defined by lifting the operators of the predicate abstract domain segment-wise.

We adapt the unification algorithm from [CCL11] in Algorithm 1 to use relational constraints from the variable environment abstract domain. The algorithm, denoted by $\textsc{unify}_{(\mathtt{I}_l, \mathtt{I}_r)}$, is parameterized by the left and right neutral element $\mathtt{I}_l$ and $\mathtt{I}_r$. The parameterization is necessary because we use the neutral elements as placeholders during unification for newly inserted segments that get later compared/joined/etc.

We give an overview of the algorithm. Note that, $\textsc{unify}_{(\mathtt{I}_l, \mathtt{I}_r)}(g, g')$ is not symmetrical with respect to its segmentation parameters g, g', since the respective predicates are preserved (as good as possible) while unifying the *limits*. The algorithm is presented as a recursive function, which can be thought as handling *tails* of the two input segmentations from a specific starting index onwards[14]. Let us call the first limit of a tail the *front limit*. The algorithm maintains the invariant[15], that the left parts up to the front limits of both segmentations are already unified.

The algorithm inspects, in each recursion, the front limits of the tails. The outermost case distinction separates cases wheter the front limits are syntactically comparable, i.e., if the bounds of one front limit are contained in the other front limit. If there are comparable, splitting the limits into parts allows ending up with syntactical equivalent front limits and move on. For the bounds not agreeing in the front limits, the inner case distinction of the algorithm decides about keeping them for later unification or drop them definitely.

If the front limits have no common bound, we use the relational comparison based on the variable environment abstract domain to know if we can still order the limits and keep the larger one for later unification. If this check does not help either, we have to drop both front limits.

[14] This indices do not appear explicitely in the algorithm but implicitely given by the first limit of the tails, L_i and L_j.

[15] The invariant is stated as a precondition in the algorithm because of the recursive definition.

Algorithm 1 Recursive Unification of $g = L_0 p_0[?]_0 L_1 \cdots$ with length n and $g' = L'_0 p'_0[?]'_0 L'_1 \cdots$ with length n' using relational constraints from the variable environment abstract domain.

$\textsc{unify}_{(\mathbb{I}_l, \mathbb{I}_r)}(L_i p_i[?]_i L_{i+1} \cdots, \quad L'_j p'_j[?]'_j L'_{j+1} \cdots)$

Precondition: The left parts $\cdots p_{i-2}[?]_{i-2} L_{i-1}$ and $\cdots p'_{i-2}[?]'_{i-2} L'_{i-1}$ are already unified

// Recursion ending criteria
if $i = n + 1 \wedge j = n' + 1$ **then**
  return

if $L_i = L'_j$ **then**
  $\textsc{unify}(L_{i+1} \cdots, \quad L'_{j+1} \cdots)$
else if $L_i \supset L'_j$ **then**
  Let B be set of bounds such that $L_i \supset B$ and $L_i - B = L_j$.
  Let $\bar{B} \subseteq B$ be set of bounds appearing later in limits of second segmentation $L'_{j+1} \cdots$
  if $\bar{B} = \varnothing$ **then**
    $\textsc{unify}(L'_j p_i[?]_i L_{i+1} \cdots, \quad L'_j p'_j[?]'_j L'_{j+1} \cdots)$
  else
    $\textsc{unify}(L'_j \; \mathbb{I}_l ? B p_i[?]_i L_{i+1} \cdots, \quad L'_j p'_j[?]'_j L'_{j+1} \cdots)$
else if $L_i \subset L'_j$ **then**
  // Symmetrical case not shown for brevity
else if $L_i \cap L'_j \neq \varnothing$ **then**
  Let $A = L_i \cap L'_j$, $B = L_i - A$, $B' = L'_j - A$.
  Let $\bar{B} \subseteq B$ be set of bounds appearing later in limits of second segmentation $L'_{j+1} \cdots$
  Let $\bar{B}' \subseteq B'$ be set of bounds appearing later in limits of first segmentation $L_{i+1} \cdots$
  if $\bar{B} = \varnothing = \bar{B}'$ **then**
    $\textsc{unify}(A p_i[?]_i L_{i+1} \cdots, \quad A p'_j[?]'_j L'_{j+1} \cdots)$
  else if $\bar{B} = \varnothing$ **then**
    $\textsc{unify}(A p_i[?]_i L_{i+1} \cdots, \quad A \; \mathbb{I}_r ? \bar{B}' p'_j[?]'_j L'_{j+1} \cdots)$
  else if $\varnothing = \bar{B}'$ **then** // Symmetrical case
    $\textsc{unify}(A \; \mathbb{I}_l ? \bar{B} p_i[?]_i L_{i+1} \cdots, \quad A p'_j[?]'_j L'_{j+1} \cdots)$
  else// $\bar{B} \neq \varnothing \wedge \bar{B}' \neq \varnothing$
    $\textsc{unify}(A \; \mathbb{I}_l ? \bar{B} p_i[?]_i L_{i+1} \cdots, \quad A \; \mathbb{I}_r ? \bar{B}' p'_j[?]'_j L'_{j+1} \cdots)$
else // $L_i \cap L'_j = \varnothing$, which implies we are not at start, i.e $i \neq 0 \wedge j \neq 0$
  if $L_i \triangleq L_j$ **then**
    $\textsc{unify}((L_i \cup L'_j) p_i[?]_i L_{i+1} \cdots, \quad (L_i \cup L'_j) p'_j[?]'_j L'_{j+1} \cdots)$
  else if $L_i \sqsubseteq L'_j$ **then**
    $\textsc{unify}(L_i p_i[?]_i L_{i+1} \cdots, \quad L_i \; \mathbb{I}_r \; L'_j p'_j[?]'_j L'_{j+1} \cdots)$
  else if $L_i \sqsupseteq L'_j$ **then** // Symmetrical case
    $\textsc{unify}(L'_j \; \mathbb{I}_r \; L_i p_i[?]_i L_{i+1} \cdots, \quad L'_j p'_j[?]'_j L'_{j+1} \cdots)$
  else
    // Remove limit L_i, L'_j, merge consecutive segments
    // If either L_i or L_j is the segmentations upper limit, do not remove it
    // (This case destinction is not shown for brevity)
    $\textsc{unify}(L_{i-1}(p_{i-1} \sqcup p_i)([?]_{i-1} \curlyvee [?]_i) L_{i+1} \cdots, \quad L'_{j-1}(p'_{j-1} \sqcup p'_i)([?]'_{j-1} \curlyvee [?]_j)' L'_{j+1} \cdots)$

Lattice Operators As mentioned, the operators are applied segment-wise after unification. For the segmentation join and widening, we use the parameterization $\text{UNIFY}_{(\bot_P,\bot_P)}$, for segmentation meet and narrowing $\text{UNIFY}_{(\top_P,\top_P)}$ and for partial order $\text{UNIFY}_{(\bot_P,\top_P)}$:

$$g \sqsubseteq_{\mathsf{Seg}} g' \stackrel{\text{def}}{=} \forall i \in \{0,\ldots,\mathsf{len}(\breve{g})-1\} : \breve{g}.p_i \sqsubseteq_P \breve{g}'.p_i \qquad \begin{array}{l}\text{where } \breve{g} = \text{UNIFY}_{(\bot_P,\top_P)}(g,g') \\ \qquad\quad \breve{g}' = \text{UNIFY}_{(\bot_P,\top_P)}(g',g)\end{array}$$

$$g \sqcup_{\mathsf{Seg}} g' \stackrel{\text{def}}{=} \breve{g}\left[p_i \leftarrow \breve{g}.p_i \sqcup_P \breve{g}'.p_i \middle| i \in \{0,\ldots,\mathsf{len}(\breve{g})-1\}\right] \qquad \begin{array}{l}\text{where } \breve{g} = \text{UNIFY}_{(\bot_P,\bot_P)}(g,g') \\ \qquad\quad \breve{g}' = \text{UNIFY}_{(\bot_P,\bot_P)}(g',g)\end{array}$$

$$g \sqcap_{\mathsf{Seg}} g' \stackrel{\text{def}}{=} \breve{g}\left[p_i \leftarrow \breve{g}.p_i \sqcap_P \breve{g}'.p_i \middle| i \in \{0,\ldots,\mathsf{len}(\breve{g})-1\}\right] \qquad \begin{array}{l}\text{where } \breve{g} = \text{UNIFY}_{(\top_P,\top_P)}(g,g') \\ \qquad\quad \breve{g}' = \text{UNIFY}_{(\top_P,\top_P)}(g',g)\end{array}$$

We add a bottom element $\bot_{\text{Seg}}$ representing an unreachable segmentation, i.e., when any of the bounds or the predicates[16] is unreachable. We do not need a special top element since we can set

$$\top_{\text{Seg}} \stackrel{\text{def}}{=} \{0\}\ \top ?\ \{list\_len\}\,.$$

Finally we get lattice $L_{\mathsf{Seg}} = \langle \mathsf{Seg}, \sqsubseteq_{\mathsf{Seg}}, \sqcup_{\mathsf{Seg}}, \sqcap_{\mathsf{Seg}}, \bot_{\mathsf{Seg}}, \top_{\mathsf{Seg}} \rangle$.

Widening The widening of segmentations is twofold. We can widen the predicates segment-wise like the other operators. This is not sufficient because it can happen that we infinitely add new limits to a segmentation. We propose a widening operator $\triangledown_{\mathsf{Seg}}$, that when applied to two segmentations $g \triangledown_{\mathsf{Seg}} g'$, drops all limits from g' that can not be found in g based on a syntactic comparison of equality.

7.2.2 Segmentation Operators

Let $\text{CLEAN}(g)$ be an operation on segmentations that merges all pairs of consecutive limits in g if they can be determined to be equal (using relational equality $\triangleq_L$). The predicate of the segment between the merged limits is discarded. Secondly, $\text{CLEAN}(g)$ removes any question mark if a segment cannot be empty, i.e., $\forall i \in \{0,\ldots,n\}$, it sets $[?]_i = \_$ if $L_i \sqsubset_L L_{i+1}$[17]. The $\text{CLEAN}(g)$ operator is introduced here for the formal presentation, in an implementation it is better performance-wise to clean up adjacent limits whenever a limit is added or changed during an operation than to do a full clean at the end of an operation.

We need a way to update a segmentation not only at specific predicate indices, but also in a range given by expressions. To find the worst-case range of affected predicates in the abstract, we introduce the *least upper limit* and *greatest lower limit* for an index given as an expression e in normal form. Let again $g = L_i p_i [?]_i L_{i+1} \cdots$ be a segmentation with length n.

$$\mathsf{luli}[g](e) \stackrel{\text{def}}{=} \operatorname*{argmin}_i \left(\{i \in \{0,\ldots,n\} | e < L_i\} \cup \{n\}\right) \qquad \text{(index of least upper limit)}$$

$$\mathsf{glli}[g](e) \stackrel{\text{def}}{=} \operatorname*{argmax}_i \left(\{i \in \{0,\ldots,n\} | L_i \leq e\} \cup \{0\}\right) \qquad \text{(index of greatest lower limit)}$$

We assume that the query expression e is always between the extremal limits, i.e., $L_0 \leq e < L_n$ (no under-/overflow of list indices). This assumption manifests in the added indices n and 0,

[16] Note that, not all predicate abstract domains include an element for representing unreachability, and even though often used as, $\bot$ does not represent an unreachable state in general.

[17] We did not define the $\sqsubset_L$, but for some combinations of expression forms and variable environments it can be defined.

which guarantee that argmin, argmax and in turn all of the following operators have a well-defined result.

Set Predicate When we want to set a predicate in the segmentation, we may need to adjust the limits to enable the most precise reflection of that predicate change. We do not only want to support updates at a constant index c, but also within an interval $[l, u]$ or at an expression e in normal form. To suit all this requirements, we define a more general function that takes two expressions e_l, e_u to bound the range where we want to set a predicate p:

$$\text{SET}_{\mathsf{Seg}}[g](e_l, e_u; p) \stackrel{\text{def}}{=} \text{CLEAN}\left(\cdots L_{\mathsf{glli}[g](e_l)} p_{\mathsf{glli}[g](e_l)} ? \{e_l\} p \{e_b + 1\} p_{\mathsf{luli}_g[g](e_u)-1} ? L_{\mathsf{luli}[g](e_u)} \cdots\right)$$

This operator creates a new segment with two new limits, the lower limit $\{e_l\}$ (inclusive) and the upper limit $\{e_b + 1\}$ (exclusive), and sets the new predicate p set between those. This segment is inserted into the segmentation between the tightest bounds such that the new segments limits conform to the order of existent limits. Uncomparable limits between those tightest bounds are removed in favor of the new segment. By the final application of the CLEAN operator the newly inserted limits might be merged with existent limits.

It is easy to overload this function to support the different ways of defining the range mentioned above, like $\text{SET}_{\mathsf{Seg}}[g](l, u; p)$, $\text{SET}_{\mathsf{Seg}}[g](c; p)$ and $\text{SET}_{\mathsf{Seg}}[g](e; p)$.

Get Predicate To query the segmentation for a predicate at a specified index, given in one of the formats listed in last paragraph, we again offer a general function to get the predicate within a query range given by e_l, e_u:

$$\text{GET}_{\mathsf{Seg}}[g](e_l, e_u) \stackrel{\text{def}}{=} \bigsqcup_P \left\{p_{\mathsf{glli}[g](e_l)}, \ldots, p_{\mathsf{luli}[g](e_u)-1}\right\}$$

This operator finds the tightest limits bounding the query range and returns the least upper bound of the predicate between those. We can again overload this function to support the different ways of defining the range, like $\text{GET}_{\mathsf{Seg}}[g](l, u)$, $\text{GET}_{\mathsf{Seg}}[g](c)$ and $\text{GET}_{\mathsf{Seg}}[g](e)$.

7.2.3 Abstract Transformers for Backward Analysis

In [CCL11] the authors present the abstract transformers for a forward analysis. For a backward analysis we need to replace the forward transformer for *variable assignment* by a backward transformer. The steps to be performed when transforming the state upon an assignment can be diveded into two parts:

1. Replace all symbolic variables used in the bounds of the segmentation.
2. Invoke the transformer of the abstract predicate domain, e.g., to reflect the flow of information (depending on the goal of the analysis). This usually involves transforming some of the predicates in the segmentation, but can also require changing the segmentation itself, e.g., adding or removing limits.

Assignment to a scalar variable (variable substitution) In an assignment $x = e$, where x is a scalar variable and e is any expression, we replace all occurences of x in the segmentation bounds by the normal form of e. If e cannot be put in normal form we have to use a fallback. The idea is to replace each bound $b = (\!|x + c|\!)$ containing x by a segment with predicate $\top_P$ covering the worst-case range $[b_l, b_u]$ that bound may take in the concrete. To achieve this, we use the variable environment abstract domain from the program point *below* the assignment to

get an interval abstraction $[e_l, e_u]$ of e, calculate $[b_l, b_u] = [e_l, e_u] + c$ and set the segmentation to $\top_P$ in that range $[b_l, b_u]$.

$$\text{REPLACE}_{\mathsf{Seg}}[\![x = e]\!]g \stackrel{\text{def}}{=} \bigsqcup_{\mathsf{Seg}} \left\{ \text{SET}_{\mathsf{Seg}}[g](b_l, b_u, \top_P) \middle| b \in \text{bounds of } g \text{ containing x} \right\}$$

Example 3. Assume we have integers variables x, y, a in a program and the variable environment provides the constraints $0 \leq a \leq 1$ and $0 \leq y$. Assume we want to transform

$$g = [\{0\}p_0?\underbrace{\{x+1\}}_{L_1}p_1?\{y\}p_2\{y+1\}p_3\{y+2\}p_4\{y+3\}p_5\{y+4\}p_6?\{\mathsf{len}(g)\}]$$

upon the assignment $[\![x = 2 * a]\!]$ with a right side that cannot be put in normal form. We evaluate $[e_l, e_u] = [0, 2]$ and thus we get the worst-case range $[b_l, b_u] = [1, 3]$ for limit L_1. Extending L_1 to the worst-case range and setting the predicate to top inside that range results in:

$$\text{REPLACE}_{\mathsf{Seg}}[\![x = 2 * a]\!]g = [\{0\}p_0\{1\}\top_P\{4\}p_5?\{y+4\}p_6?\{\mathsf{len}(g)\}]$$

Implementation-wise, instead of joining all the individual segmentations reflecting a change in one bound each, we can use an iterative application of $\text{SET}_{\mathsf{Seg}}$. Note that, if e contains any input() expressions, this leads to an interval abstraction of the right side of $[b_l, b_u] = [-\infty, \infty]$. In that case, all inner limits have to be to be dropped. However, the way $\text{SET}_{\mathsf{Seg}}$ is defined, it ensures the extremal limits are never removed.

Assignment to a list In a list assignment $l = l'$, where l is a list variable and l' is either another list variable or a list display expression, we do not have to replace anything inside a segmentation since segmentations do not contain list variables. So we only perform step 2.

In an assignment $l[t] = e$ to a list index, where l is a list variable and t is a scalar expression and e any expression, we again only have to perform step 2.

7.3 List Segmentation for Usage Property

We instantiate the generic list segmentation abstract domain for the usage analysis to get $\mathbf{USeg} = \mathbf{Seg}(\mathbf{L}(\mathbf{E_{Oct}}), \mathbf{U}, \mathbf{Oct})$. We use the octagon domain as the variable environement abstract domain and the expressions of the form as in Section 7.1.2 using octagonal constraints for the rich comparison operators.

7.3.1 Integer/List stores

We track the scalar (integer) variables of the program with members of the usage domain **U** as in the integer-only programs from Section 5.2, and we track list variables with segmentations. In **USeg** we use abstract stores $q \in Q$, $q = q_{\text{int}} \cup q_{\text{list}}$, where $q_{\text{int}} : \mathbb{V}\text{ar}_{\text{int}} \to \mathcal{U}$ and $q_{\text{list}} : \mathbb{V}\text{ar}_{\text{list}} \to \mathsf{USeg}$.

Information Flow Operators We lift the function REPLACE to stores:

$$\text{REPLACE}_Q[\![x = e]\!]q \stackrel{\text{def}}{=} q\left[l \leftarrow \text{REPLACE}_{\mathsf{Seg}}[\![x = e]\!]q(l) \mid l \in \mathbb{V}\text{ar}_{\text{list}}\right]$$

For other types of assignment this operator is the identity.

We extend usage stores to be able to filter expressions containing both scalar variable accessses and list index expressions. We define the condition when the operands in a filtered expression are considered used, i.e., when a store is *dirty*, as its own operator:

$$\begin{aligned}\textsc{dirty}(q) \stackrel{\text{def}}{=} &\exists y \in \mathbb{V}\mathrm{ar}_{\text{int}} : q(y) \in \{\downarrow, \nleftarrow\} \\ &\vee \exists l \in \mathbb{V}\mathrm{ar}_{\text{list}} : \exists i \in \{0, \ldots, \mathsf{len}(q(l)) - 1\} : q(l).p_i \in \{\downarrow, \nleftarrow\}\end{aligned}$$

Implicit Flow We account for implicit flow by checking if a store is dirty and updating the store accordingly:

$$\begin{aligned}\textsc{filter}_Q[\![e]\!]q \stackrel{\text{def}}{=}\ & q\left[x \xleftarrow{\textsc{dirty}(q)} \downarrow \middle| x \in \mathsf{opd}(e) \cap \mathbb{V}\mathrm{ar}_{\text{int}}\right] \\ & \dot{\sqcup}\ q\left[l \xleftarrow{\textsc{dirty}(q)} \textsc{set}_{\mathsf{Seg}}[q(l)](t; \downarrow) \middle| l[t] \in \mathsf{opd}(e) \cap \mathbb{I}\mathrm{ndex}\right]\end{aligned}$$

Explicit Flow We account for explicit flow by checking if left side of assignment is currently used:

$$\begin{aligned}\textsc{use}_Q[\![x = e]\!]q \stackrel{\text{def}}{=}\ & q\left[y \xleftarrow{q(x) \in \{\downarrow, \downarrow\}} \downarrow \middle| y \in \mathsf{opd}(e) \cap \mathbb{V}\mathrm{ar}_{\text{int}}\right] \\ & \dot{\sqcup}\ q\left[l \xleftarrow{q(x) \in \{\downarrow, \downarrow\}} \textsc{set}_{\mathsf{Seg}}[q(l)](t; \downarrow) \middle| l[t] \in \mathsf{opd}(e) \cap \mathbb{I}\mathrm{ndex}\right]\end{aligned}$$

The transformer $\textsc{use}_Q[\![l[t] = e]\!]q$ is similar except that the condition when to update the store is $\textsc{get}_{\mathsf{Seg}}[q(l)](t) \in \{\downarrow, \downarrow\}$. When assigning to a list variable, we propagate the used property naturally to the right side of the assignment, is it another list variable or a list display expression:

$$\textsc{use}_Q[\![l = l']\!]q \stackrel{\text{def}}{=} q\left[l' \leftarrow q(l)\left[p_i \xleftarrow{l \neq l' \wedge p_i \in \{\downarrow, \downarrow\}} \downarrow \middle| i \in \{0, \ldots, \mathsf{len}(l)\}\right]\right]$$

$$\textsc{use}_Q[\![l = [t_0, \ldots, t_{m-1}]]\!]q \stackrel{\text{def}}{=} q\left[x \xleftarrow{\textsc{get}_{\mathsf{Seg}}[q(l)](k) \in \{\downarrow, \downarrow\}} \downarrow \middle| x \in \mathsf{opd}(t_k) \cap \mathbb{V}\mathrm{ar}_{\text{int}}, k \in \{0, \ldots, m-1\}\right]$$

Intermitted Flow The intermitted flow for an assignment to a scalar variable is as before:

$$\textsc{kill}_Q[\![x = e]\!]q \stackrel{\text{def}}{=}\ q\left[x \xleftarrow{q(x) \in \{\downarrow, \downarrow\} \wedge x \notin \mathsf{opd}(e)} \nleftarrow\right]$$

We add the definitions for the assignment to list variables:

$$\textsc{kill}_Q[\![l = l']\!]q \stackrel{\text{def}}{=} q\left[l \leftarrow q(l)\left[p_i \xleftarrow{l \neq l' \wedge p_i \in \{\downarrow, \downarrow\}} \nleftarrow \middle| i \in \{0, \ldots, \mathsf{len}(l)\}\right]\right]$$

$$\textsc{kill}_Q[\![l = [t_0, \ldots, t_{m-1}]]\!]q \stackrel{\text{def}}{=} q\left[l \leftarrow q(l)\left[p_i \xleftarrow{p_i \in \{\downarrow, \downarrow\}} \nleftarrow \middle| i \in \{0, \ldots, \mathsf{len}(l)\}\right]\right]$$

To determine the intermitted flow for an assignment to a list index $l[t] = e$, we have to do a little more work. We have to find the subset of indices of t that do not appear in the expression

e on the right side:

$$\text{KILL}_Q[\![l[t] = e]\!]q \stackrel{\text{def}}{=} q[l \leftarrow$$
$$q(l)\left[\begin{array}{l} p_i \xleftarrow{p_i \in \{\downarrow,\downarrow\}} \lightning \end{array}\middle|\begin{array}{l} i \in \underbrace{\{\mathsf{glli}[q(l)](t), \ldots, \mathsf{luli}[q(l)](t) - 1\}}_{\text{predicate indices overwritten left}} \\ \quad - \underbrace{\bigcup\left\{ \{\mathsf{glli}[q(l)](r), \ldots, \mathsf{luli}[q(l)](r) - 1\} \middle| l[r] \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{list}} \right\}}_{\text{predicate indices used right}} \end{array}\right]$$
$$]$$

Finally we can combine this operators to adapt the partial abstract semantics given in (3) for int/list stores and the backwards substitution required by segmentations:

$$\begin{aligned} [\![\psi = \omega]\!]q &\stackrel{\text{def}}{=} \left(\text{REPLACE}_Q[\![\psi = \omega]\!] \circ \text{KILL}_Q[\![\psi = \omega]\!] \circ \text{USE}_Q[\![\psi = \omega]\!]\right) q \\ [\![\text{print}(e)]\!]q &\stackrel{\text{def}}{=} q\left[x \leftarrow \downarrow \middle| x \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{int}}\right] \\ &\quad \dot{\sqcup}\, q\left[l \leftarrow \text{SET}_{\mathsf{Seg}}[q(l)](t; \downarrow) \middle| l[t] \in \mathsf{opd}(e) \cap \mathbb{I}\text{ndex}\right] \\ &\quad \dot{\sqcup}\, q\left[l \leftarrow \top_{\mathsf{Seg}} \middle| l \in \mathsf{opd}(e) \cap \mathbb{V}\text{ar}_{\text{list}}\right] \end{aligned}$$

7.3.2 Abstract Transformers

A notable difference to the transformers for the analysis for int-only programs is that upon an assignment, we have to transform all stack frames. In the top frame we have to call the complete transformer $[\![\psi = \omega]\!]$ as before (which includes the variable replacement and update of usage). In all other frames except the the top frame we have to only replace the assigned variable with REPLACE_Q.

$$\begin{aligned} [\![\textbf{pass}]\!]s &\stackrel{\text{def}}{=} s \\ [\![\psi = \omega]\!]s &\stackrel{\text{def}}{=} \langle \text{REPLACE}_Q[\![\psi = \omega]\!]s_1, \ldots, \text{REPLACE}_Q[\![\psi = \omega]\!]s_{k-1}, [\![\psi = \omega]\!]s_k \rangle \\ [\![\text{print}(e)]\!]s &\stackrel{\text{def}}{=} \langle s_1, \ldots, s_{k-1}, [\![\text{print}(e)]\!]s_k \rangle \\ [\![\textbf{if } b : stmt_1 \textbf{ else } stmt_2]\!]s &\stackrel{\text{def}}{=} \text{POP}((\text{FILTER}_S[\![b]\!] \circ [\![stmt_1]\!])(\text{PUSH}(s))) \\ &\quad \bar{\sqcup}\, \text{POP}((\text{FILTER}_S[\![\textbf{ not } b]\!] \circ [\![stmt_2]\!])(\text{PUSH}(s))) \\ [\![\textbf{while } b : stmt]\!]s &\stackrel{\text{def}}{=} \mathsf{lfp}_s\left[\lambda t.\text{FILTER}_S[\![\textbf{ not } b]\!]t \,\bar{\sqcup}\, \text{POP}((\text{FILTER}_S[\![b]\!] \circ [\![stmt]\!])(\text{PUSH}(t)))\right] \end{aligned}$$

7.3.3 Concretization

The generic concretization given in [CCL11] is fully applicable to our instantiation, since we did not change parts of the segmentation relevant for the concretization.

7.3.4 Example

We inspect the two bugs in Code 1 from the introduction separately. Code 4a shows a variant with only Bug A in place. Since Bug A makes the whole while-loop effectless on the output, the analysis correctly detects on line 4:

$$count \to \downarrow,\ i \to \bot,\ g \to \bot,\ list\_grades \to [\{0\}\, \bot?\{\mathsf{len}(list\_grades)\}]$$

Code 4b shows a variant with only Bug B in place. The while-loop now has an effect, but the first item in the list is excluded from the counting. Thus, the result on line 4 indicates that the list is only used from index 1 onwards:

$$count \to \downarrow,\ i \to \downarrow,\ g \to \lightning,$$
$$list\_grades \to [\{0\}\, \bot?\{i\}\, \downarrow\{i+1\}\, \downarrow\{i+1\}\, \downarrow\{i+1\}\, \downarrow?\{\mathsf{len}(list\_grades)\}]$$

```
1  list_grades = listinput()

   count = 0

5  i = 0
   while i < len(list_grades):
       g = list_grades[i]
       if g >= 4:
           g += 1  # Bug A
10
       i += 1

   print(count)
```

(a) with only Bug A

```
1  list_grades = listinput()

   count = 0

5  i = 1  # Bug B
   while i < len(list_grades):
       g = list_grades[i]
       if g >= 4:
           count += 1
10
       i += 1

   print(count)
```

(b) with only Bug B

Code 4: Programs calculating the number of passing grades, each with a different bug.

8 Implementation

All domains described in this report have been implemented and tested in the abstract interpretation framework *Lyra*[18], which was built alongside this project as a generic framework to explore abstract interpretation analysis.

8.1 Framework for Abstract Interpretation

The framework is written in Python v3.5. It follows a generic approach to support different input languages by translating the input programs into an intermediate control flow graph (CFG) representation. The CFG contains links to the program points the statements derive their origin from. The interpreter[19] transforms the abstract states while traversing the CFG and collects the analysis results. How the states are transformed can be specified by customizable semantics. The states itself are built of a hierarchy of abstract domains or partial domains. A state offers only a narrow interface to the interpreter and does not have any knowledge about the CFG structure but knows how to transform itself upon assignments, conditions etc.

Lattice and BoundedLattice At the core of each domain is at least one lattice structure. In case a domain is used directly by an interpreter, it must built on a lattice that has a dedicated bottom element. This is used by the interpreter to represent unreachability.

State A state is itself a lattice, combining one or several lattices to an abstract domain, handling the interaction between them and offering all necessary interface methods for the interpreter, i.e., the abstract transformers.

Generic Domains The abstract domain functors from Section 4 have their direct equivalent in the framework, called *Generic Domains*. They are designed as superclasses or mixins to generalize frequent structures of lattices like the variable store.

8.2 Automated Testing

While not every analysis supports all constructs of a modern programming language, it is still helpful to limit the input programs to a real subset of, e.g., Python, to make the sample programs actually runnable.

Apart from usual unit tests, we added a full-chain testing framework. This operates on files, each representing a test, and does all the steps necessary to test the analysis on the code in the file:

1. Parse the file into the abstract syntax tree (AST),
2. translate the AST to the CFG,
3. run one or multiple connected analysis on it,
4. check the actual analysis results against expected results provided as special comments in the input files,

[18] https://github.com/caterinaurban/Lyra

[19] The interpreter of the analysis must not be confused with the Python interpreter.

5. plot (intermediate) results for manual inspection.

8.3 Examples

Code 5 shows an erroneous example program. The programmers intention was to halve a given integer x n-times via integer (floor) division. The bug is marked on line 6. The analysis correctly detects on line 4 that i and n (and therefore the complete body of the loop) are effectless. Figure 2 shows the CFG for this program with all intermediate results. If we correct the bug, the analysis result on line 4 becomes $n \to \downarrow$ (and $i \to \not\downarrow$) as expected.

```
1  n = int(input())
   x = int(input())
   i = 0
   # RESULT:  i↛⊥,  n↛⊥,  x↛⊥
5  while i < n:
       x = i / 2  # Bug: should be x = x / 2
       i = i + 1
   else:
       x = -1
10 print(x)
```

Code 5: Program demonstrating expected result specification.

A second example is shown in Code 6. For input grades from 1 (worst) to 6 (best), the program calculates the points a student gained with core and minor subjects throughout a semester. Points are awarded for every grade above 4 (line 15 and 30), and deducted for grades below 4. Grades of core subjects below 4 are subtraced twice (line 17).[20] There are 3 bugs in the code. Bug A causes *total_point* not be updated with the points accumulated with core subjects. Note that all variables are *truly-live* according to the definition in the introduction. Our analysis is able to detect that the variable *points* is unused at line 15 and line 17, thus the variable $diff$ and g are also unused in the first while loop. Finally the analysis correctly detects that none of the values in the list of core grades are used at line 3:

$$list\_core\_grades \to [\{0\}\, \bot?\{\mathsf{len}(list\_core\_grades)\}]$$

Bug B and bug C causes the first and the last element of the list of minor grades to be unused. While our algorithm is able to detect Bug B as demonstrated in the example in Section 7.3.4, Bug C is not found because the widening operator is applied in the loop header node after some iterations. This results in the segmentation abstraction for the list of minor grades at line 3:

$$list\_minor\_grades \to [\{0\}\, \bot?\{i\}\, \downarrow\{i+1\}\, \downarrow\{i+2\}\, \downarrow\{i+3\}\, \downarrow?\{\mathsf{len}(list\_minor\_grades)\}]$$

This result could be improved by a more sophisticated widening strategy. There is research that tries to improve widening generically under reasonable performance losses [GR06].

[20] The double deducting of grades below 4 is not made up, this was the rule at the authors secondary school.

```
list_core_grades = list(map(int, input().split()))
list_minor_grades = list(map(int, input().split()))

# Total accumulated points
points = 0
total_points = 0

# CALCULATE CORE SUBJECTS POINTS
i = 0
while i < len(list_core_grades):
    g = list_core_grades[i]

    diff = g - 4
    if diff >= 0:  # passing grade
        points += diff
    else:  # diff < 0
        points += 2 * diff

    i += 1

# Bug A: missing 'total_points += points'
points = 0

# CALCULATE MINOR SUBJECTS POINTS
i = 1  # Bug B: should be 'i = 0'
while i < len(list_core_grades) - 1:  # Bug C: should be <=
    g = list_minor_grades[i]

    diff = g - 4
    points += diff

    i += 1

total_points += points

if total_points >= 0:
    result = 1  # passed
else:
    result = -1  # failed

print(result)
```

Code 6: Program collecting points of core and minor subjects and determining if a student passed the semester.

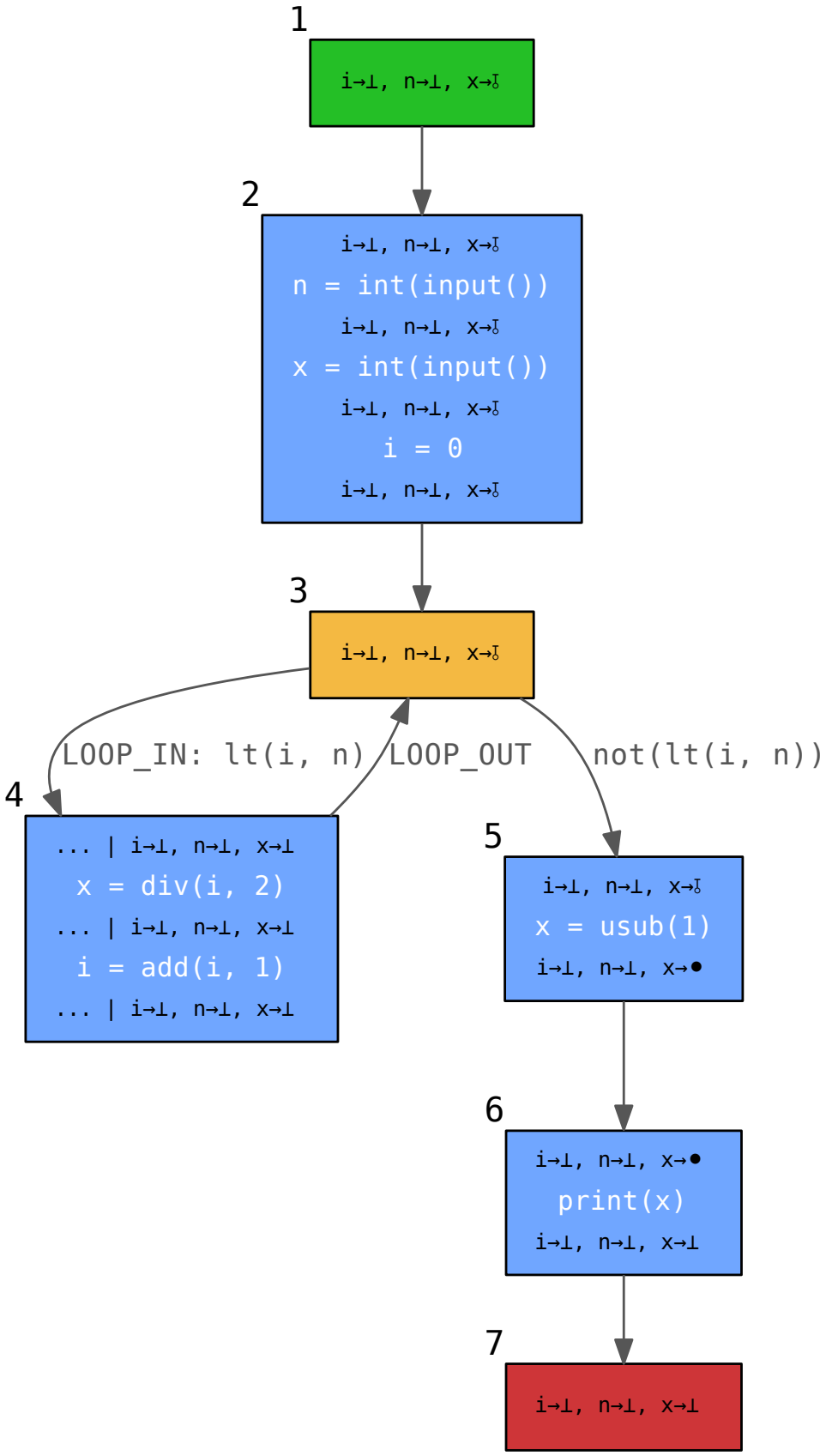


Figure 2: The CFG with usage analysis results for Code 5.

9 Conclusion

In this thesis, we started off from well studied problems involving properties of variables of a program, such as the property of *live* and *truly-live* variables and corresponding analysis. We motivated the need for a stronger property, which describes if an input or a variable is used directly, transitively or implicitely by an output statement. We developed a notion and formal definition for the new usage property.

In the framework of Abstract Interpretation, we constructed a simple but powerful abstract domain that approximates the usage property for scalar variables with a backwards analysis. We showed that the concept can be extended to lists, using two approaches: A simple lifting of the usage property to the start of a list already allows detecting bugs involving the content of lists, but the precision is not satisfying. As a next step, we built on known concepts to track list contents by segmentations and instantiated these for the usage property. In this process, we optimized segmentations for our property and improved some of the required operators like the unification of two segmentations.

The prototype developed alongside this thesis implements all the theoretically derived analysis for a simplified Python language and shows their potential in real-world applications. Because the analysis finds a property at the level of variables, we could easily extend it to local variables in functions or methods and parameters of those. The analysis could then be used as an ad-hoc analyzer in programming IDEs.

Future Work

Possible improvements are twofold: First, we could refine the usage property itself to preserve more detail about how an input or variable is used for the analysis report, e.g., if it is used explicitely or implicitely (in a condition). We could even go as far as finding *how often* or *how much* an input is used and detect irregularities when inputs are expected to contribute equally to the result. Additionally, we could improve the precision of the presented analysis. In the list content usage analysis, the widening operator can be improved to drop limits of a segmentation more carefully. This would allow finding bugs where we miss items at the end of a list.

Secondly, the analysis support can be extended to other language constructs, such as sets, nested lists, procedures and classes. The latter two would ask for an interprocedural analysis. The prototype implementation could be made more user-friendly by reporting back analysis results only for inputs. Because sometimes a program intentionally skips inputs unused, we could either introduce a way for the user to specify which inputs should be checked for the usage property, or we add a pre-analysis that tries to detect this by heuristics.

References

[BDR04] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[BHZ08] Roberto Bagnara, Patricia Hill, and Enea Zaffanella. An improved tight closure algorithm for integer octagonal constraints. In *Verification, Model Checking, and Abstract Interpretation*, pages 8–21. Springer, 2008.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*, volume 46, pages 105–118. ACM, 2011.

[GM81] Robert Giegerich and Ulrich Möncke. Invariance of approximative semantics with respect to program transformations. In *GI11. Jahrestagung*, pages 1–10. Springer, 1981.

[GR06] Denis Gopan and Thomas W Reps. Lookahead widening. In *CAV*, volume 4144, pages 452–466. Springer, 2006.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[Urb15] Caterina Urban. *Static analysis by abstract interpretation of functional temporal properties of programs.* PhD thesis, École Normale Supérieure, 2015.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Static Program Analysis of Data Usage Properties

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Wehrli	Simon

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 27.8.2017	S. Wehrli

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.