An Abstract Interpretation Framework for Input Data Usage

Caterina Urban and Peter Müller

Department of Computer Science, ETH Zurich, Switzerland
{caterina.urban, peter.mueller}@inf.ethz.ch

Abstract. Data science software plays an increasingly important role in critical decision making in fields ranging from economy and finance to biology and medicine. As a result, errors in data science applications can have severe consequences, especially when they lead to results that look plausible, but are incorrect. A common cause of such errors is when applications erroneously ignore some of their input data, for instance due to bugs in the code that reads, filters, or clusters it.
In this paper, we propose an abstract interpretation framework to automatically detect unused input data. We derive a program semantics that precisely captures data usage by abstraction of the program's operational trace semantics and express it in a constructive fixpoint form. Based on this semantics, we systematically derive static analyses that automatically detect unused input data by fixpoint approximation.
This clear design principle provides a framework that subsumes existing analyses; we show that secure information flow analyses and a form of live variables analysis can be used for data usage, with varying degrees of precision. Additionally, we derive a static analysis to detect single unused data inputs, which is similar to dependency analyses used in the context of backward program slicing. Finally, we demonstrate the value of expressing such analyses as abstract interpretation by combining them with an existing abstraction of compound data structures such as arrays and lists to detect unused chunks of the data.

1 Introduction

In the past few years, data science has grown considerably in importance and now heavily influences many domains, ranging from economy and finance to biology and medicine. As we rely more and more on data science for making decisions, we become increasingly vulnerable to programming errors.

Programming errors can cause frustration, especially when they lead to a program failure after hours of computation. However, programming errors that do not cause failures can have more serious consequences as code that produces an erroneous but plausible result gives no indication that something went wrong. A notable example is the paper "Growth in a Time of Debt" published in 2010 by economists Reinhart and Rogoff, which was widely cited in political debates and was later demonstrated to be flawed. Notably, one of the flaws was a programming error, which *entirely excluded some data* from the analysis [23]. Its critics

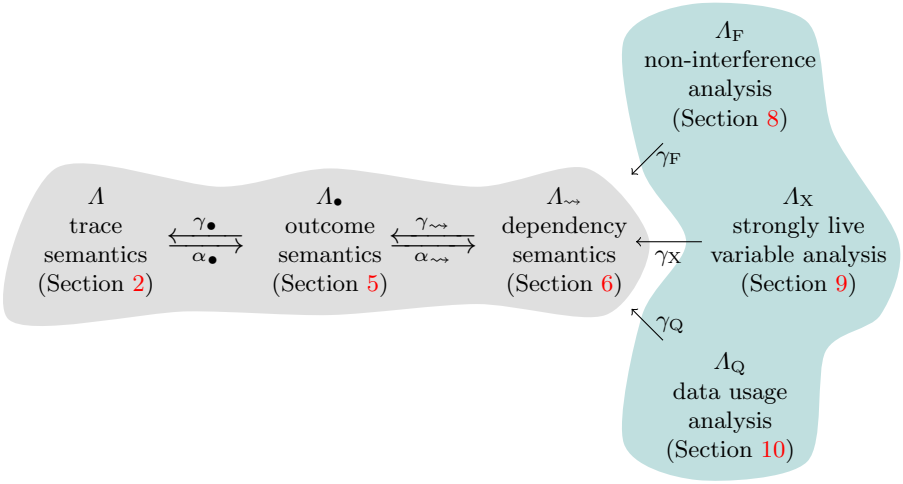

Fig. 1. Overview of the program semantics presented in the paper. The *dependency semantics*, derived by abstraction of the *trace semantics*, is sound and complete for data usage. Further sound but not complete abstractions are shown on the right.

hold that this paper led to unjustified adoption of austerity policies for countries with various levels of public debt [30]. Programming errors in data analysis code for medical applications are even more critical [27]. It is thus paramount to achieve a high level of confidence in the correctness of data science code.

The likelihood that a programming error causes some input data to remain unused is particularly high for data science applications, where data goes through long pipelines of modules that acquire, filter, merge, and manipulate it. In this paper, we propose an abstract interpretation [14] framework to automatically detect *unused input data*. We characterize when a program uses (some of) its input data using the notion of *dependency* between the input data and the *outcome* of the program. Our notion of dependency accounts for non-determinism and non-termination. Thus, it encompasses notions of dependency that arise in many different contexts, such as secure information flow and program slicing [1], as well as provenance or lineage analysis [9], to name a few.

Following the theory of abstract interpretation [12], we systematically derive a new program semantics that precisely captures exactly the information needed to reason about input data usage, abstracting away from irrelevant details about the program behavior. Figure 1 gives an overview of our approach. The semantics is first expressed in a constructive fixpoint form over *sets of sets of traces*, by partitioning the operational trace semantics of a program based on its outcome (cf. *outcome semantics* in Figure 1), and a further abstraction ignores intermediate state computations (cf. *dependency semantics* in Figure 1). Starting the development of the semantics from the operational trace semantics enables a uniform mathematical reasoning about programs semantics and program properties (Section 3). In particular, since input data usage is not a trace property

or a subset-closed property [11] (Section 4), we show that a formulation of the semantics using sets of sets of traces is necessary for a sound validation of input data usage via fixpoint approximation [28].

This clear design principle provides a unifying framework for reasoning about existing analyses based on dependencies. We survey existing analyses and identify key design decisions that limit or facilitate their applicability to input data usage, and we assess their precision. We show that non-interference analyses [6] are sound for proving that a *terminating* program does not use *any* of its input data; although this is too strong a property in general. We prove that strongly live variable analysis [20] is sound for data usage even for non-terminating programs, albeit it is imprecise with respect to implicit dependencies between program variables. We then derive a more precise static analysis similar to dependency analyses used in the context of backward program slicing [37]. Finally, we demonstrate the value of expressing these analyses as abstract interpretations by combining them with an existing abstraction of compound data structures such as arrays and lists [16]. This allows us to detect unused chunks of the input data, and thus apply our work to realistic data science applications.

2 Trace Semantics

The *semantics* of a program is a mathematical characterization of its behavior when executed for all possible input data. We model the operational semantics of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states [12,14]. Note that this model allows representing programs with (possibly unbounded) non-determinism. The set $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau\}$ is the set of *final states* of the program.

In the following, let $\Sigma^n \stackrel{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in \Sigma\}$ be the set of all sequences of exactly n program states. We write ε to denote the empty sequence, i.e., $\Sigma^0 \stackrel{\text{def}}{=} \{\varepsilon\}$. Let $\Sigma^\star \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \Sigma^n$ be the set of all finite sequences, $\Sigma^+ \stackrel{\text{def}}{=} \Sigma^\star \setminus \Sigma^0$ be the set of all non-empty finite sequences, Σ^ω be the set of all infinite sequences, $\Sigma^{+\infty} \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$ be the set of all non-empty finite or infinite sequences and $\Sigma^{\star\infty} \stackrel{\text{def}}{=} \Sigma^\star \cup \Sigma^\omega$ be the set of all finite or infinite sequences of program states. In the following, we write $\sigma\sigma'$ for the concatenation of two sequences $\sigma, \sigma' \in \Sigma^{\star\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ when $\sigma \in \Sigma^\omega$), $T^+ \stackrel{\text{def}}{=} T \cap \Sigma^+$ and $T^\omega \stackrel{\text{def}}{=} T \cap \Sigma^\omega$ for the selection of the non-empty finite sequences and the infinite sequences of $T \in \mathcal{P}(\Sigma^{\star\infty})$, and $T \,;\, T' \stackrel{\text{def}}{=} \{\sigma s \sigma' \mid s \in \Sigma \wedge \sigma s \in T \wedge s\sigma' \in T'\}$ for the merging of two sets of sequences $T \in \mathcal{P}(\Sigma^+)$ and $T' \in \mathcal{P}(\Sigma^{+\infty})$, when a finite sequence in T terminates with the initial state of a sequence in T'.

Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of program states described by the transition relation τ, that is, $\langle s, s' \rangle \in \tau$ for each pair of consecutive states $s, s' \in \Sigma$ in the sequence. The set of final states Ω and the transition relation τ can be understood as sets of traces of length one and length two, respectively. The *trace semantics* $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition

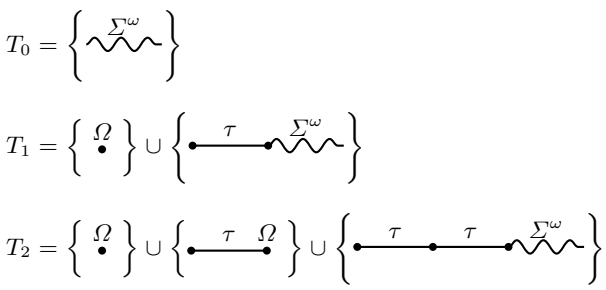

Fig. 2. First fixpoint iterates of the trace semantics Λ.

system $\langle \Sigma, \tau \rangle$ is the union of all finite traces that are terminating with a final state in Ω, and all infinite traces. It can be expressed as a least fixpoint in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ [12]:

$$\begin{aligned} \Lambda &= \mathrm{lfp}^{\sqsubseteq}\, \Theta \\ \Theta(T) &\stackrel{\text{def}}{=} \Omega \cup (\tau \,;\, T) \end{aligned} \tag{1}$$

where the computational order is $T_1 \sqsubseteq T_2 \stackrel{\text{def}}{=} T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$. Figure 2 illustrates the first fixpoint iterates. The fixpoint iteration starts from the set of all infinite *sequences* of program states. At each iteration, the final program states in Ω are added to the set, and sequences already in the set are extended by prepending transitions to them. In this way, we *add* increasingly longer finite traces, and we *remove* infinite sequences of states with increasingly longer prefixes not forming traces. In particular, the i-th iterate builds all finite traces of length less than or equal to i, and selects all infinite sequences whose prefixes of length i form traces. At the limit we obtain all infinite traces and all finite traces that terminate in a final state in Ω. Note that Λ is *suffix-closed*.

The trace semantics Λ fully describes the behavior of a program. However, to reason about a particular property of a program, it is not necessary to consider all aspects of its behavior. In fact, reasoning is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In the next sections, we define our property of interest and use abstract interpretation [14] to systematically derive, by successive abstractions of the trace semantics, a semantics that precisely captures such a property.

3 Input Data Usage

A *property* is specified by its extension, that is, the set of elements having such a property [14,15]. Thus, properties of program traces in $\Sigma^{+\infty}$ are sets of traces in $\mathcal{P}(\Sigma^{+\infty})$, and properties of programs with trace semantics in $\mathcal{P}(\Sigma^{+\infty})$ are *sets of sets of traces* in $\mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$. Accordingly, a program P satisfies a property $\mathcal{H} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ if and only if its semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$ belongs to $\mathcal{H}$:

$$P \models \mathcal{H} \Leftrightarrow [\![P]\!] \in \mathcal{H} \tag{2}$$

Some program properties are defined in terms of individual program traces and can be equivalently expressed as trace properties. This is the case for the traditional safety [26] and liveness [4] properties of programs. In such a case, a program P satisfies a trace property $\mathcal{T}$ if and only if all traces in its semantics $[\![P]\!]$ belong to the property: $P \models \mathcal{T} \Leftrightarrow [\![P]\!] \subseteq \mathcal{T}$.

Program properties that establish a relation between different program traces cannot be expressed as trace properties [11]. Examples are security properties such as *non-interference* [21,35]. In this paper, we consider a closely related but more general property called *input data usage*, which expresses that *the outcome of a program does not depend on (some of) its input data*. The notion of *outcome* accounts for non-determinism as well as non-termination. Thus, our notion of dependency encompasses non-interference as well as notions of dependency that arise in many other contexts [1,9]. We further explore this in Sections 8 to 10.

Let each program P with trace semantics $[\![P]\!]$ have a set I_P of input variables and a set O_P of output variables[1]. For simplicity, we can assume that these variables are all of the same type (e.g., boolean variables) and their values are all in a set V of possible values (e.g., $\mathrm{V} = \{\mathtt{T}, \mathtt{F}\}$ where $\mathtt{T}$ is the boolean value true and $\mathtt{F}$ is the boolean value false). Given a trace $\sigma \in [\![P]\!]$, we write $\sigma[0]$ to denote its initial state and $\sigma[\omega]$ to denote its outcome, that is, its final state if the trace is finite or $\perp$ if the trace is infinite. The input variables at the initial states of the traces of a program store the values of its input data: we write $\sigma[0](i)$ to denote the value of the input data stored in the input variable i at the initial state of the trace σ, and $\sigma_1[0] \neq_i \sigma_2[0]$ to denote that the initial states of two traces σ_1 and σ_2 disagree on the value of the input variable i but agree on the values of all other variables. The output variables at the final states of the finite traces of a program store its result: we write $\sigma[\omega](o)$ to denote the result stored in the output variable o at the final state of a finite trace σ. We can now formally define when an input variable $i \in \mathrm{I}_P$ is *unused* with respect to a program with trace semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$:

$$\begin{aligned} \textsc{unused}_i([\![P]\!]) \stackrel{\text{def}}{=} \forall \sigma \in [\![P]\!], v \in \mathrm{V} \colon \sigma[0](i) \neq v \Rightarrow \\ \exists \sigma' \in [\![P]\!] \colon \sigma'[0] \neq_i \sigma[0] \wedge \sigma'[0](i) = v \wedge \sigma[\omega] = \sigma'[\omega] \end{aligned} \tag{3}$$

Intuitively, an input variable i is unused if all feasible program outcomes (e.g., the outcome $\sigma[\omega]$ of a trace σ) are feasible from all possible initial values of i (i.e., for all possible initial values v of i that differ from the initial value of i in σ, there exists a trace with initial value v for i that has the same outcome $\sigma[\omega]$). In other words, the outcome of the program is the same independently of the initial value of the input variable i. Note that this definition accounts for non-determinism (since it considers each program outcome independently) and non-termination (since a program outcome can be $\perp$).

Example 1. Let us consider the simple program P in Figure 3. Based on the input variables `english`, `math`, and `science` (cf. lines 1-3), the program is supposed to

[1] The approach can be easily extended to infinite inputs and/or outputs via abstractions such as the one later presented in Section 11.

```
1  english = input()
2  math = input()
3  science = input()
4  bonus = input()
5
6  passing = True
7  if not english: english = False              # english should be passing
8  if not math: passing = bonus
9  if not math: passing = bonus                 # math should be science
10
11 print(passing)
```

Fig. 3. Simple program to check if a student has passed three school subjects. The programmer has made two mistakes at line 7 and at line 9, which cause the input data stored in the variables `english` and `science` to be unused.

check if a student has passed all three considered school subjects and store the result in the output variable `passing` (cf. line 11). For mathematics and science, the student is allowed a bonus based on the input variable `bonus` (cf. line 8 and 9). However, the programmer has made two mistakes at line 7 and at line 9, which cause the input variables `english` and `science` to be unused.

Let us now consider the input variable `science`. The trace semantics of the program (simplified to consider only the variables `science` and `passing`) is:

$$[\![P]\!]_{\texttt{science}} = \{(\mathrm{T}\_)\dots(\mathrm{TT}), (\mathrm{T}\_)\dots(\mathrm{TF}), (\mathrm{F}\_)\dots(\mathrm{FT}), (\mathrm{F}\_)\dots(\mathrm{FF})\}$$

where each state $(v_1 v_2)$ shows the boolean value v_1 of `science` and v_2 of `passing`, and _ denotes any boolean value. We omitted the trace suffixes for brevity. The input variable `science` is *unused*, since each result value (T or F) for `passing` is feasible from all possible initial values of `science`. Note that all other outcomes of the program (i.e., non-termination) are not feasible.

Let us now consider the input variable `math`. The trace semantics of the program (now simplified to only consider `math` and `passing`) is the following:

$$[\![P]\!]_{\texttt{math}} = \{(\mathrm{T}\_)\dots(\mathrm{TT}), (\mathrm{F}\_)\dots(\mathrm{FT}), (\mathrm{F}\_)\dots(\mathrm{FF})\}$$

In this case, the input variable `math` is used since only the initial state (F_) yields the result value F for `passing` (in the final state (FF)). ■

The input data usage property $\mathcal{N}$ can now be formally defined as follows:

$$\mathcal{N} \stackrel{\text{def}}{=} \left\{ [\![P]\!] \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \forall i \in \mathrm{I}_P : \textsc{unused}_i([\![P]\!]) \right\} \tag{4}$$

which states that the outcome of a program does not depend on *any* input data. In practice one is interested in weaker input data usage properties for a subset J of the input variables, i.e., $\mathcal{N}_J \stackrel{\text{def}}{=} \{ [\![P]\!] \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \forall i \in J \subseteq \mathrm{I}_P : \textsc{unused}_i([\![P]\!]) \}$.

In the following, we use abstract interpretation to reason about input data usage. In the next section, we discuss the challenges to the application of the standard abstract interpretation framework that emerge from the fact that input data usage cannot be expressed as a trace property.

4 Sound Input Data Usage Validation

In the standard framework of abstract interpretation, one defines a semantics that precisely captures a property $\mathcal{S}$ of interest by abstraction of the trace semantics Λ [12]. Then, further abstractions $\Lambda^\sharp$ provide sound over-approximations $\gamma(\Lambda^\sharp)$ of Λ (by means of a concretization function γ): $\Lambda \subseteq \gamma(\Lambda^\sharp)$. For a *trace property*, an over-approximation $\gamma([\![P]\!]^\sharp)$ of the semantics $[\![P]\!]$ of a program P allows a sound validation of the property: since $[\![P]\!] \subseteq \gamma([\![P]\!]^\sharp)$, we have that $\gamma([\![P]\!]^\sharp) \subseteq \mathcal{S} \Rightarrow [\![P]\!] \subseteq \mathcal{S}$ and so, if $\gamma([\![P]\!]^\sharp) \subseteq \mathcal{S}$, we can conclude that $P \models \mathcal{S}$ (cf. Section 3). This conclusion is also valid for all other *subset-closed* properties [11]: since by definition $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \Rightarrow \forall T \subseteq \gamma([\![P]\!]^\sharp) \colon T \in \mathcal{S}$, we have that $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \Rightarrow [\![P]\!] \in \mathcal{S}$ (and so we can conclude that $P \models \mathcal{S}$). However, for program properties that are not subset-closed, we have that $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \not\Rightarrow [\![P]\!] \in \mathcal{S}$ [28] and so we cannot conclude that $P \models \mathcal{S}$, even if $\gamma([\![P]\!]^\sharp) \in \mathcal{S}$ (cf. Equation 2).

We have seen in the previous section that input data usage is not a trace property. The example below shows that it is *not* a subset-closed property either.

Example 2. Let us consider again the program P and its semantics $[\![P]\!]_{\texttt{science}}$ and $[\![P]\!]_{\texttt{math}}$ shown in Example 1. We have seen in Example 1 that the semantics $[\![P]\!]_{\texttt{science}}$ belongs to the data usage property $\mathcal{N}$: $[\![P]\!]_{\texttt{science}} \in \mathcal{N}$. Let us consider now the following subset T of $[\![P]\!]_{\texttt{science}}$:

$$T = \{(\mathrm{T}\_) \dots (\mathrm{TT}), (\mathrm{F}\_) \dots (\mathrm{FT}), (\mathrm{F}\_) \dots (\mathrm{FF})\}$$

In this case, the input variable `science` is used. Indeed, we can observe that T coincides with $[\![P]\!]_{\texttt{math}}$ (except for the considered input variable). Thus $T \notin \mathcal{N}$ even though $T \subseteq [\![P]\!]_{\texttt{science}}$. ■

Since input data usage is not subset-closed, we are in the unfortunate situation that we cannot use the standard abstract interpretation framework to soundly prove that a program does not use (some of) its input data using an over-approximation of the semantics of the program: $\gamma([\![P]\!]^\sharp) \in \mathcal{N}_J \not\Rightarrow [\![P]\!] \in \mathcal{N}_J$.

We solve this problem in the next section, by lifting the trace semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$ of a program P (i.e., a set of traces) to a set of sets of traces $(\!|P|\!) \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ [28]. In this setting, a program P satisfies a property $\mathcal{H}$ if and only if its semantics $(\!|P|\!)$ is a subset of $\mathcal{H}$:

$$P \models \mathcal{H} \Leftrightarrow (\!|P|\!) \subseteq \mathcal{H} \tag{5}$$

As we will explain in the next section, now an over-approximation $\gamma((\!|P|\!)^\sharp)$ of $(\!|P|\!)$ allows again a sound validation of the property: since $(\!|P|\!) \subseteq \gamma((\!|P|\!)^\sharp)$, we have that $\gamma((\!|P|\!)^\sharp) \subseteq \mathcal{H} \Rightarrow (\!|P|\!) \subseteq \mathcal{H}$ (and so we can conclude that $P \models \mathcal{H}$).

More specifically, in the next section, we define a program semantics $(\!|P|\!)$ that precisely captures which subset J of the input variables is unused by a program P. In later sections, we present further abstractions $(\!|P|\!)^\sharp$ that over-approximate the subset of the input variables that *may be used* by P, and thus allows a sound validation of an *under-approximation* $J^\sharp$ of J: $\gamma((\!|P|\!)^\sharp) \subseteq \mathcal{N}_{J^\sharp} \Rightarrow (\!|P|\!) \subseteq \mathcal{N}_{J^\sharp}$. In other words, this means that every input variable reported as unused by an abstraction is indeed not used by the program.

5 Outcome Semantics

We lift the trace semantics Λ to a set of sets of traces by *partitioning*. The *partitioning abstraction* $\alpha_Q \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ of a set of traces T is:

$$\alpha_Q(T) \stackrel{\text{def}}{=} \{T \cap C \mid C \in Q\} \tag{6}$$

where $Q \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ is a *partition* of sequences of program states.

More specifically, to reason about input data usage of a program P, we lift the trace semantics $[\![P]\!]$ to $(\!|P|\!)$ by partitioning it into sets of traces that yield the same program outcome. The key insight behind this idea is that, given an input variable i, the initial states of all traces in a partition give all initial values for i that yield a program outcome; the variable i is unused if and only if these initial values are all the possible values for i (or the set of values is empty because the outcome is unfeasible, cf. Equation 3). Thus, if the trace semantics $[\![P]\!]$ of a program P belongs to the input data usage property $\mathcal{N}_J$, then each partition in $(\!|P|\!)$ must also belong to $\mathcal{N}_J$, and vice versa: we have that $[\![P]\!] \in \mathcal{N}_J \Leftrightarrow (\!|P|\!) \subseteq \mathcal{N}_J$, which is precisely what we want (cf. Equation 5).

Let $T^+_{o=v}$ denote the subset of the finite sequences of program states in $T \in \mathcal{P}(\Sigma^{+\infty})$ with value v for the output variable o in their outcome (i.e., their final state): $T^+_{o=v} \stackrel{\text{def}}{=} \{\sigma \in T^+ \mid \sigma[\omega](o) = v\}$. We define the *outcome partition* $O \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ of sequences of program states:

$$O \stackrel{\text{def}}{=} \left\{ \Sigma^+_{o_1=v_1,\dots,o_k=v_k} \mid v_1, \dots, v_k \in \mathrm{V} \right\} \cup \{\Sigma^\omega\}$$

where V is the set of possible values of the output variables $o_1, \dots, o_k$ (cf. Section 3). The partition contains all sets of finite sequences that agree on the values of the output variables in their outcome, and all infinite sequences of program states (i.e., all sequences with outcome $\bot$). We instantiate α_Q above with the outcome partition to obtain the *outcome abstraction* $\alpha_\bullet \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$:

$$\alpha_\bullet(T) \stackrel{\text{def}}{=} \left\{ T^+_{o_1=v_1,\dots,o_k=v_k} \mid v_1, \dots, v_k \in \mathrm{V} \right\} \cup \{T^\omega\} \tag{7}$$

Example 3. The program P of Example 1 has only one output variable `passing` with boolean value T or F. Let us consider again the trace semantics $[\![P]\!]_{\texttt{math}}$ shown in Example 1. Its outcome abstraction $\alpha_\bullet([\![P]\!]_{\texttt{math}})$ is:

$$\alpha_\bullet([\![P]\!]_{\texttt{math}}) = \{\emptyset, \{(\mathrm{F}\_) \dots (\mathrm{FF})\}, \{(\mathrm{T}\_) \dots (\mathrm{TT}), (\mathrm{F}\_) \dots (\mathrm{FT})\}\}$$

Note that all traces with different result values for the output variable `passing` belong to different sets of traces (i.e., partitions) in $\alpha_\bullet([\![P]\!]_{\texttt{math}})$. The empty set corresponds to the (unfeasible) non-terminating outcome of the program. ■

We can now use the outcome abstraction $\alpha_\bullet$ to define the *outcome semantics* $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ as an abstraction of the trace semantics Λ:

Definition 1. *The* outcome semantics $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ *is defined as:*

$$\Lambda_\bullet \stackrel{def}{=} \alpha_\bullet(\Lambda) \tag{8}$$

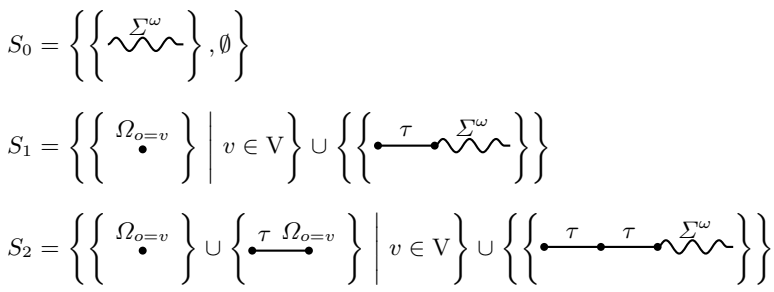

Fig. 4. First iterates of the outcome semantics $\Lambda_\bullet$ for a single output variable o.

where $\alpha_\bullet$ is the outcome abstraction (cf. Equation 7) and $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ is the trace semantics (cf. Equation 1).

The outcome semantics contains the set of all infinite traces and all sets of finite traces that agree on the value of the output variables in their outcome.

In the following, we express the outcome semantics $\Lambda_\bullet$ in a constructive fixpoint form. This allows us to later derive further abstractions of $\Lambda_\bullet$ by *fixpoint transfer* and *fixpoint approximation* [12]. Given a set of sets of traces S, we write $S^+_{o=v} \stackrel{\text{def}}{=} \{T \in S \mid T = T^+_{o=v}\}$ for the selection of the sets of traces in S that agree on the value v of the output variable o in their outcome, and $S^\omega \stackrel{\text{def}}{=} \{T \in S \mid T = T^\omega\}$ for the selection of the sets of infinite traces in S. When $S^+_{o=v}$ (resp. S^ω) contains a single set of traces T, we abuse notation and write $S^+_{o=v}$ (resp. S^ω) to also denote T. The following result gives a fixpoint definition of $\Lambda_\bullet$ in the complete lattice $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \boxdot, \sqcup, \sqcap, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$, where the computational order $\boxdot$ is defined (similarly to $\sqsubseteq$, cf. Section 2) as:

$$S_1 \boxdot S_2 \stackrel{\text{def}}{=} \bigwedge_{v_1,\ldots,v_k \in \mathrm{V}} {S_1}^+_{o_1=v_1,\ldots,o_k=v_k} \subseteq {S_2}^+_{o_1=v_1,\ldots,o_k=v_k} \wedge S_1^\omega \supseteq S_2^\omega$$

Theorem 1. *The outcome semantics $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ can be expressed as a least fixpoint in $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \boxdot, \sqcup, \sqcap, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$ as:*

$$\begin{aligned} \Lambda_\bullet &= \mathrm{lfp}^{\boxdot}\, \Theta_\bullet \\ \Theta_\bullet(S) &\stackrel{def}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1,\ldots,v_k \in \mathrm{V}\} \uplus \{\tau\,;T \mid T \in S\} \end{aligned} \tag{9}$$

where $S_1 \uplus S_2 \stackrel{def}{=} \left\{ {S_1}^+_{o_1=v_1,\ldots,o_k=v_k} \cup {S_2}^+_{o_1=v_1,\ldots,o_k=v_k} \mid v_1,\ldots,v_k \in \mathrm{V} \right\} \cup S_1^\omega \cup S_2^\omega$.

Figure 4 illustrates the first fixpoint iterates of the outcome semantics for a single output variable o. The fixpoint iteration starts from the partition containing the set of all infinite sequences of program states and the empty set (which represents an empty set of finite traces). At the first iteration, the empty set is replaced with a partition of the final states Ω based on the value v of the output variable o, while the infinite sequences are extended by prepending transitions to them (similarly to the trace semantics, cf. Equation 1). At the next iterations, all

sequences contained in each partition are further extended, and the final states that agree on the value v of o are again added to the matching set of traces that agree on v in their outcome. At the limit, we obtain a partition containing the set of all infinite traces and all sets of finite traces that agree on the value v of the output variable o in their outcome.

To prove Theorem 1 we first need to show that the outcome abstraction $\alpha_\bullet$ preserves least upper bounds of non-empty sets of sets of traces.

Lemma 1. *The outcome abstraction $\alpha_\bullet$ is Scott-continuous.*

Proof. We need to show that for any non-empty ascending chain C of sets of traces with least upper bound $\sqcup C$, we have that $\alpha_\bullet(\sqcup C) = \dot{\bigcup}\{\alpha_\bullet(T) \mid T \in C\}$, that is, $\alpha_\bullet(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$, the image of C via $\alpha_\bullet$.

First, we know that $\alpha_\bullet$ is monotonic, i.e., for any two sets of traces T_1 and T_2 we have $T_1 \sqsubseteq T_2 \Rightarrow \alpha_\bullet(T_1) \mathrel{\dot{\sqsubseteq}} \alpha_\bullet(T_2)$. Since $\sqcup C$ is the least upper bound of C, for any set T in C we have that $T \sqsubseteq \sqcup C$ and, since $\alpha_\bullet$ is monotonic, we have that $\alpha_\bullet(T) \mathrel{\dot{\sqsubseteq}} \alpha_\bullet(\sqcup C)$. Thus $\alpha(\sqcup C)$ is an upper bound of $\{\alpha_\bullet(T) \mid T \in C\}$.

To show that $\alpha(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$, we need to show that for any other upper bound U of $\alpha_\bullet(C)$ we have $\alpha_\bullet(\sqcup C) \mathrel{\dot{\sqsubseteq}} U$. Let us assume by absurd that $\alpha_\bullet(\sqcup C) \not\mathrel{\dot{\sqsubseteq}} U$. Then, there exists $T_1 \in \alpha_\bullet(\sqcup C)$ and $T_2 \in U$ such that $T_1 \not\sqsubseteq T_2$: $T_1^+ \supset T_2^+$ or $T_1^\omega \subset T_2^\omega$. Let us assume that $T_1^+ \supset T_2^+$. By definition of $\alpha_\bullet$, we observe that T_1 is a partition of $\dot{\bigcup} C$ and, since $\dot{\bigcup} C$ is the least upper bound of C, U cannot be an upper bound of $\alpha_\bullet(C)$ (since T_2 does not contain enough finite traces). Similarly, if $T_1^\omega \subset T_2^\omega$, then U cannot be an upper bound of $\alpha_\bullet(C)$ (since T_2 contains too many infinite traces). Thus, we must have $\alpha_\bullet(\sqcup C) \mathrel{\dot{\sqsubseteq}} U$ and we can conclude that $\alpha(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$. □

We can now prove Theorem 1 by Kleenian fixpoint transfer [12].

Proof (Sketch). The proof follows by Kleenian fixpoint transfer. We have that $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \mathrel{\dot{\sqsubseteq}}, \dot{\bigcup}, \dot{\bigcap}, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$ is a complete lattice and that $\phi^{+\infty}$ (cf. Equation 1) and $\Theta_\bullet$ (cf. Equation 8) are monotonic function. Additionally, we have that the outcome abstraction $\alpha_\bullet$ (cf. Equation 7) is Scott-continuous (cf. Lemma 1) and such that $\alpha_\bullet(\Sigma^\omega) = \{\Sigma^\omega, \emptyset\}$ and $\alpha_\bullet \circ \phi^{+\infty} = \Theta_\bullet \circ \alpha_\bullet$. Then, by Kleenian fixpoint transfer, we have that $\alpha_\bullet(\Lambda) = \alpha_\bullet(\mathrm{lfp}^{\sqsubseteq}\ \phi^{+\infty}) = \mathrm{lfp}^{\mathrel{\dot{\sqsubseteq}}}\ \Theta_\bullet$. Thus, we can conclude that $\Lambda_\bullet = \mathrm{lfp}^{\mathrel{\dot{\sqsubseteq}}}\ \Theta_\bullet$. □

Finally, we show that the outcome semantics $\Lambda_\bullet$ is sound and complete for proving that a program does not use (a subset of) its input variables.

Theorem 2. *A program does not use a subset J of its input variables if and only if its outcome semantics $\Lambda_\bullet$ is a subset of $\mathcal{N}_J$:*

$$P \models \mathcal{N}_J \Leftrightarrow \Lambda_\bullet \subseteq \mathcal{N}_J$$

Proof (Sketch). The proof follows immediately from the definition of $\mathcal{N}_J$ (cf. Equation 3 and Section 4) and the definition of $\Lambda_\bullet$ (cf. Equation 8). □

Example 4. Let us consider again the program P and its semantics $[\![P]\!]_{\texttt{science}}$ shown in Example 1. The corresponding outcome semantics $\alpha_\bullet([\![P]\!]_{\texttt{science}})$ is:

$$\alpha_\bullet([\![P]\!]_{\texttt{science}}) = \{\emptyset, \{(\texttt{T\_}) \dots (\texttt{TF}), (\texttt{F\_}) \dots (\texttt{FF})\}, \{(\texttt{T\_}) \dots (\texttt{TT}), (\texttt{F\_}) \dots (\texttt{FT})\}\}$$

Note that all sets of traces in $\alpha_\bullet([\![P]\!]_{\texttt{science}})$ belong to $\mathcal{N}_{\{\texttt{science}\}}$: the initial states of all traces in a non-empty partition contain all possible initial values (T or F) for the input variable `science`. Thus, P satisfies $\mathcal{N}_{\{\texttt{science}\}}$ and, indeed, the input variable `science` is unused by P. ■

As discussed in Section 4, we now can again use the standard framework of abstract interpretation to soundly over-approximate $\Lambda_\bullet$ and prove that a program does not use (some of) its input data. In the next section, we propose an abstraction that remains sound and complete for input data usage. Further sound but not complete abstractions are presented in later sections.

6 Dependency Semantics

We observe that, to reason about input data usage, it is not necessary to consider all intermediate state computations between the initial state of a trace and its outcome. Thus, we can further abstract the outcome semantics $\Lambda_\bullet$ into a set $\Lambda_{\rightsquigarrow}$ of (dependency) relations between initial states and outcomes of a set of traces.

We lift the abstraction defined for this purpose on sets of traces [12] to $\alpha_{\rightsquigarrow} : \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\bot))$ on sets of sets of traces:

$$\alpha_{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \{\{\langle \sigma[0], \sigma[\omega]\rangle \in \Sigma \times \Sigma_\bot \mid \sigma \in T\} \mid T \in S\} \tag{10}$$

where $\Sigma_\bot \stackrel{\text{def}}{=} \Sigma \cup \{\bot\}$. The *dependency abstraction* $\alpha_{\rightsquigarrow}$ ignores all intermediate states between the initial state $\sigma[0]$ and the outcome $\sigma[\omega]$ of all traces σ in all partitions T of S. Observe that a trace σ that consists of a single state s is abstracted as a pair $\langle s, s\rangle$. The corresponding dependency concretization function $\gamma_{\rightsquigarrow} : \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\bot)) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ over-approximates the original sets of traces by inserting arbitrary intermediate states:

$$\gamma_{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \left\{T \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \{\langle \sigma[0], \sigma[\omega]\rangle \in \Sigma \times \Sigma_\bot \mid \sigma \in T\} \in S\right\} \tag{11}$$

Example 5. Let us consider again the program of Example 1 and its outcome semantics $\alpha_\bullet([\![P]\!]_{\texttt{math}})$ shown in Example 3. Its dependency abstraction is:

$$\alpha_{\rightsquigarrow}(\alpha_\bullet([\![P]\!]_{\texttt{math}})) = \{\emptyset, \{\langle \texttt{F\_}, \texttt{FF}\rangle\}, \{\langle \texttt{T\_}, \texttt{TT}\rangle, \langle \texttt{F\_}, \texttt{FT}\rangle\}\}$$

which explicitly ignores intermediate program states. ■

Using $\alpha_{\rightsquigarrow}$, we now define the *dependency semantics* $\Lambda_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ as an abstraction of the outcome semantics $\Lambda_\bullet$.

Definition 2. *The* dependency semantics $\Lambda_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ *is defined as:*

$$\Lambda_{\rightsquigarrow} \stackrel{def}{=} \alpha_{\rightsquigarrow}(\Lambda_{\bullet}) \tag{12}$$

where $\Lambda_{\bullet} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ *is the outcome semantics (cf. Equation 8) and* $\alpha_{\rightsquigarrow}$ *is the dependency abstraction (cf. Equation 10).*

Neither the Kleenian fixpoint transfer nor the Tarskian fixpoint transfer can be used to obtain a fixpoint definition for the dependency semantics, but we have to proceed by union of disjoint fixpoints [12]. To this end, we observe that the outcome semantics $\Lambda_{\bullet}$ can be equivalently expressed as follows:

$$\begin{aligned} \Lambda_{\bullet} &= \Lambda^{+}_{\bullet} \cup \Lambda^{\omega}_{\bullet} = \mathrm{lfp}^{\sqsubseteq}_{\emptyset}\, \Theta^{+}_{\bullet} \cup \mathrm{lfp}^{\sqsubseteq}_{\{\Sigma^{\omega}\}}\, \Theta^{\omega}_{\bullet} \\ \Theta^{+}_{\bullet}(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1,\ldots,v_k \in \mathrm{V}\} \uplus \{\tau\,;T \mid T \in S\} \\ \Theta^{\omega}_{\bullet}(S) &\stackrel{\text{def}}{=} \{\tau\,;T \mid T \in S\} \end{aligned} \tag{13}$$

where $\Lambda^{+}_{\bullet}$ and $\Lambda^{\omega}_{\bullet}$ separately compute the set of all sets of finite traces that agree on their outcome, and the set of all infinite traces, respectively.

In the following, given a set of traces $T \in \mathcal{P}(\Sigma^{+\infty})$ and its dependency abstraction $\alpha_{\rightsquigarrow}(T)$, we abuse notation and write T^{+} (resp. T^{ω}) to also denote $\alpha_{\rightsquigarrow}(T)^{+} \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(T) \cap (\Sigma \times \Sigma)$ (resp. $\alpha_{\rightsquigarrow}(T)^{\omega} \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(T) \cap (\Sigma \times \{\bot\})$). Similarly, we reuse the symbols for the computational order $\sqsubseteq$, least upper bound $\sqcup$, and greatest lower bound $\sqcap$, instead of their abstractions. We can now use the Kleenian and Tarskian fixpoint transfer to separately derive fixpoint definitions of $\alpha_{\rightsquigarrow}(\Lambda^{+}_{\bullet})$ and $\alpha_{\rightsquigarrow}(\Lambda^{\omega}_{\bullet})$ in $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\bot})), \sqsubseteq, \sqcup, \sqcap, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$.

Lemma 2. *The abstraction* $\Lambda^{+}_{\rightsquigarrow} \stackrel{def}{=} \alpha_{\rightsquigarrow}(\Lambda^{+}_{\bullet}) \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\bot})), \sqsubseteq, \sqcup, \sqcap, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda^{+}_{\rightsquigarrow} &= \mathrm{lfp}^{\sqsubseteq}_{\{\emptyset\}}\, \Theta^{+}_{\rightsquigarrow} \\ \Theta^{+}_{\rightsquigarrow}(S) &\stackrel{def}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \times \Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1,\ldots,v_k \in \mathrm{V}\} \uplus \{\tau \circ R \mid R \in S\} \end{aligned} \tag{14}$$

Proof (Sketch). By Kleenian fixpoint transfer (cf. Theorem 17 in [12]). □

Lemma 3. *The abstraction* $\Lambda^{\omega}_{\rightsquigarrow} \stackrel{def}{=} \alpha_{\rightsquigarrow}(\Lambda^{\omega}_{\bullet}) \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\bot})), \sqsubseteq, \sqcup, \sqcap, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda^{\omega}_{\rightsquigarrow} &= \mathrm{lfp}^{\sqsubseteq}_{\{\Sigma\times\{\bot\}\}}\, \Theta^{\omega}_{\rightsquigarrow} \\ \Theta^{\omega}_{\rightsquigarrow}(S) &\stackrel{def}{=} \{\tau \circ R \mid R \in S\} \end{aligned} \tag{15}$$

Proof (Sketch). By Tarskian fixpoint transfer (cf. Theorem 18 in [12]). □

The fixpoint iteration for $\Lambda^{+}_{\rightsquigarrow}$ starts from the set containing only the empty relation. At the first iteration, the empty relation is replaced by all relations between pairs of final states that agree on the values of the output variables.

At each next iteration, all relations are combined with the transition relation to obtain relations between initial and final states of increasingly longer traces. At the limit, we obtain the set of all relations between the initial and the final states of a program that agree on the final value of the output variables. The fixpoint iteration for $\Lambda^{\omega}_{\rightsquigarrow}$ starts from the set containing (the set of) all pairs of states and the $\bot$ outcome, and each iteration discards more and more pairs with initial states that do not belong to infinite traces of the program.

Now we can use Lemma 2 and Lemma 3 to express the dependency semantics $\Lambda_{\rightsquigarrow}$ in a constructive fixpoint form (as the union of $\Lambda^{+}_{\rightsquigarrow}$ and $\Lambda^{\omega}_{\rightsquigarrow}$).

Theorem 3. *The dependency semantics $\Lambda_{\rightsquigarrow} \in \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma_{\bot}\right)\right)$ can be expressed as a least fixpoint in $\langle \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma_{\bot}\right)\right), \sqsubseteq^{\cdot}, \dot{\cup}, \dot{\cap}, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ as:*

$$\Lambda_{\rightsquigarrow} = \Lambda^{+}_{\rightsquigarrow} \cup \Lambda^{\omega}_{\rightsquigarrow} = \mathrm{lfp}^{\sqsubseteq^{\cdot}}_{\{\Sigma\times\{\bot\},\emptyset\}}\ \Theta_{\rightsquigarrow}$$

$$\Theta_{\rightsquigarrow}(S) \stackrel{def}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \times \Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1,\ldots,v_k \in \mathrm{V}\} \dot{\cup} \{\tau \circ R \mid R \in S\} \tag{16}$$

Proof (Sketch). The proof follows immediately from Lemma 2 and Lemma 3. □

Finally, we show that the dependency semantics $\Lambda_{\rightsquigarrow}$ is sound and complete for proving that a program does not use (a subset of) its input variables.

Theorem 4. *A program does not use a subset J of its input variables if and only if the image via $\gamma_{\rightsquigarrow}$ of its dependency semantics $\Lambda_{\rightsquigarrow}$ is a subset of $\mathcal{N}_J$:*

$$P \models \mathcal{N}_J \Leftrightarrow \gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$$

Proof (Sketch). The proof follows from the definition of $\Lambda_{\rightsquigarrow}$ (cf. Equation 12) and $\gamma_{\rightsquigarrow}$ (cf. Equation 11), and from Theorem 2. □

Example 6. Let us consider again the program P and its outcome semantics $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$ from Example 4. The corresponding dependency semantics is:

$$\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})) = \{\emptyset, \{\langle \mathrm{T}\_, \mathrm{TF}\rangle, \langle \mathrm{F}\_, \mathrm{FF}\rangle\}, \{\langle \mathrm{T}\_, \mathrm{TT}\rangle, \langle \mathrm{F}\_, \mathrm{FT}\rangle\}\}$$

and, by definition of $\gamma_{\rightsquigarrow}$, we have that its concretization $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ is an over-approximation of $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$. In particular, since intermediate state computations are irrelevant for deciding the input data usage property, all sets of traces in $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ are over-approximations of exactly one set in $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$ with the same set of initial states and outcome. Thus, in this case, we can observe that all sets of traces in $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ belong to $\mathcal{N}_{\{\texttt{science}\}}$ and correctly conclude that P does not use the variable `science`. ■

At this point we have a sound and complete program semantics that captures only the minimal information needed to decide which input variables are unused by a program. In the rest of the paper, we present various static analyses for input data usage by means of sound abstractions of this semantics, which *under-approximate* (resp. over-approximate) the subset of the input variables that are *unused* (resp. used) by a program.

7 Input Data Usage Abstractions

We introduce a simple sequential programming language with boolean variables, which we use for illustration throughout the rest of the paper:

$$e ::= v \mid x \mid \texttt{not}\ e \mid e\ \texttt{and}\ e \mid e\ \texttt{or}\ e \qquad \text{(expressions)}$$
$$s ::= \texttt{skip} \mid x = e \mid \texttt{if}\ e\texttt{:}\ s\ \texttt{else:}\ s \mid \texttt{while}\ e\texttt{:}\ s \mid s\ s \qquad \text{(statements)}$$

where v ranges over boolean values, and x ranges over program variables. The `skip` statement, which does nothing, is a placeholder useful, for instance, for writing a conditional `if` statement without an `else` branch: `if` e`:` s `else: skip`. In the following, we often simply write `if` e`:` s instead of `if` e`:` s `else: skip`. Note that our work is not limited by the choice of a particular programming language, as the formal treatment in previous sections is language independent.

In Section 8 and 9, we show that existing static analyses based on dependencies [6,20] are abstractions of the dependency semantics $\Lambda_{\rightsquigarrow}$. We define each abstraction $\Lambda^\sharp$ over a partially ordered set $\langle \mathcal{A}, \sqsubseteq_A \rangle$ called *abstract domain*. More specifically, for each program statement s, we define a *transfer function* $\Theta^\sharp[\![s]\!] : \mathcal{A} \to \mathcal{A}$, and the abstraction $\Lambda^\sharp$ is the composition of the transfer functions of all statements in a program. We derive a more precise static analysis similar to dependency analyses used for program slicing [37] in Section 10. Finally, Section 11 demonstrates the value of expressing such analyses as abstract domains by combining them with an existing abstraction of compound data structures such as arrays and lists [16] to detect unused chunks of input data.

8 Secure Information Flow Abstractions

Secure information flow analysis [18] aims at proving that a program will not leak sensitive information. Most analyses focus on proving *non-interference* [35] by classifying program variables into different security levels [17], and ensuring the absence of information flow from variables with higher security level to variables with lower security level. The most basic classification comprises a low security level L, and a high security level H: program variables classified as L are public information, while variables classified as H are private information.

In our context, if we classify input variables as H and all other variables as L, possiblistic non-interference [21] coincides with the input data usage property $\mathcal{N}$ (cf. Equation 4) *restricted to consider only terminating programs*. However, in general, (possibilistic) non-interference is too strong for our purposes as it requires that *none* of the input variables is used by a program. We illustrate this using as an example a non-interference analysis recently proposed by Assaf et al. [6] that is conveniently formalized in the framework of abstract interpretation. We briefly present here a version of the originally proposed analysis, simplified to consider only the security levels L and H, and we point out the significance of the definitions for input data usage.

Let $\mathcal{L} \stackrel{\text{def}}{=} \{L, H\}$ be the set of security levels, and let the set X of all program variables be partitioned into a set X_L of variables classified as L and a set X_H

of variables classified as H (i.e., the input variables). A dependency constraint $L \rightsquigarrow x$ expresses that the current value of the variable x depends only on the initial values of variables having at most security level L (i.e., it does not depend on the initial value of any of the input variables). The non-interference analysis Λ_{F} proposed by Assaf et al. is a *forward analysis* in the lattice $\langle \mathcal{P}(\mathrm{F}), \sqsubseteq_{\mathrm{F}}, \sqcup_{\mathrm{F}} \rangle$ where $\mathrm{F} \stackrel{\text{def}}{=} \{L \rightsquigarrow x \mid x \in \mathrm{X}\}$ is the set of all dependency constraints, $S_1 \sqsubseteq_{\mathrm{F}} S_2 \stackrel{\text{def}}{=} S_1 \supseteq S_2$, and $S_1 \sqcup_{\mathrm{F}} S_2 \stackrel{\text{def}}{=} S_1 \cap S_2$. The transfer function $\Theta_{\mathrm{F}}[\![s]\!] : \mathcal{P}(\mathrm{F}) \to \mathcal{P}(\mathrm{F})$ for each statement s in our simple programming language is defined as follows:

$$
\begin{aligned}
\Theta_{\mathrm{F}}[\![\texttt{skip}]\!](S) &\stackrel{\text{def}}{=} S \\
\Theta_{\mathrm{F}}[\![x = e]\!](S) &\stackrel{\text{def}}{=} \{L \rightsquigarrow y \in S \mid y \neq x\} \cup \{L \rightsquigarrow x \mid \mathcal{V}_{\mathrm{F}}[\![e]\!]S\} \\
\Theta_{\mathrm{F}}[\![\texttt{if } e\texttt{: } s_1 \texttt{ else: } s_2]\!](S) &\stackrel{\text{def}}{=} \begin{cases} \Theta_{\mathrm{F}}[\![s_1]\!](S) \sqcup_{\mathrm{F}} \Theta_{\mathrm{F}}[\![s_2]\!](S) & \text{if } \mathcal{V}_{\mathrm{F}}[\![e]\!]S \\ \{L \rightsquigarrow x \in S \mid x \notin \mathrm{W}(s_1) \cup \mathrm{W}(s_2)\} & \text{otherwise} \end{cases} \\
\Theta_{\mathrm{F}}[\![\texttt{while } e\texttt{: } s]\!](S) &\stackrel{\text{def}}{=} \mathrm{lfp}_S^{\sqsubseteq_{\mathrm{F}}}\ \Theta_{\mathrm{F}}[\![\texttt{if } e\texttt{: } s \texttt{ else: skip}]\!] \\
\Theta_{\mathrm{F}}[\![s_1\ s_2]\!](S) &\stackrel{\text{def}}{=} \Theta_{\mathrm{F}}[\![s_2]\!] \circ \Theta_{\mathrm{F}}[\![s_1]\!](S)
\end{aligned}
$$

where $\mathrm{W}(s)$ denotes the set of variables modified by the statement s, and $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ determines whether a set of dependencies S guarantees that the expression e has a unique value independently of the initial value of the input variables. For a variable x, $\mathcal{V}_{\mathrm{F}}[\![x]\!]S$ is true if and only if $L \rightsquigarrow x \in S$. Otherwise, $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ is defined recursively on the structure of e, and it is always true for a boolean value v [6]. An assignment $x = e$ discards all dependency constraints related to the assigned variable x, and adds constraints $L \rightsquigarrow x$ if e has a unique value independently of the initial values of the input variables. This captures an *explicit flow* of information between e and x. A conditional statement `if` e`:` s_1 `else:` s_2 joins the dependency constraints obtained from s_1 and s_2, if e does not depend on the initial values of the input variables (i.e., $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ is true). Otherwise, it discards all dependency constraints related to the variables modified in either of its branches. This captures an *implicit flow* of information from e. The initial set of dependencies contains a constraint $L \rightsquigarrow x$ for each variable x that is not an input variable. We exemplify the analysis below.

Example 7. Let us consider again the program P from Example 1 (stripped of the `input` and `print` statements, which are not present in our simple language):

```
1 passing = True
2 if not english: english = False              # english should be passing
3 if not math: passing = bonus
4 if not math: passing = bonus                 # math should be science
```

The analysis begins from the set of dependency constraints $\{L \rightsquigarrow \texttt{passing}\}$, which classifies input variables as H and all other variables as L. The assignment at line 1 leaves the set unchanged as the value of the expression `True` on the right-hand side of the assignment does not depend on the initial value of the input variables. The set remains unchanged by the conditional statement at line 2, even though the boolean condition depends on the input variable `english`,

because the variable `passing` is not modified. Finally, at line 3 and 4, the analysis captures an explicit flow of information from the input variable `bonus` and an implicit flow of information from the input variable `math`. Thus, the set of dependency constraints becomes empty at line 3, and remains empty at line 4.

Observe that, in this case, non-interference does not hold since the result of the program depends on some of the input variables. Therefore, the analysis is only able to conclude that at least one of the input variables may be used by the program, but it cannot determine which input variables are unused. ■

The example shows that non-interference is too strong a property in general. Of course, one could determine which input variables are unused by running multiple instances of the non-interference analysis Λ_{F}, each one of them classifying a single different input variable as H and all other variables as L. However, this becomes cumbersome in a data science application where a program reads and manipulates a large amount of input data.

Moreover, we emphasize that our input data usage property is more general than (possibilistic) non-interference since it also considers non-termination. We are not aware of any work on termination-sensitive possibilistic non-interference.

Example 8. Let us modify the program P shown in Example 7 as follows:

```
1 passing = True
2 while not english: english = False
```

In this case, since the loop at line 2 does not modify the output variable `passing`, the non-interference analysis Λ_{F} will leave the initial set of dependency constraints $\{L \rightsquigarrow \texttt{passing}\}$ unchanged, meaning that the result of the program does not depend on any of its input variables. However, the input variable `english` is used since its value influences the outcome of the program: the program terminates if `english` is true, and does not terminate otherwise. ■

The example demonstrates that the analysis is *unsound* for a non-terminating program.[2] We show that the non-interference analysis Λ_{F} is sound for proving that a program does not use any of its input variables, *only if the program is terminating*. We define the concretization function $\gamma_{\mathrm{F}} \colon \mathcal{P}(\mathrm{F}) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$:

$$\gamma_{\mathrm{F}}(S) \stackrel{\text{def}}{=} \{R \in \mathcal{P}(\Sigma \times \Sigma) \mid \alpha_{\mathrm{F}}(R) \sqsubseteq_{\mathrm{F}} S\} \tag{17}$$

The abstraction function $\alpha_{\mathrm{F}} \colon \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma)) \to \mathcal{P}(\mathrm{F})$ maps each relation R between states of a program to the corresponding set of dependency constraints: $\alpha_{\mathrm{F}}(R) \stackrel{\text{def}}{=} \{L \rightsquigarrow x \mid x \in \mathrm{X}_L \wedge \forall i \in \mathrm{X}_H \colon \textsc{unused}_{i,x}(R)\}$, where $\textsc{unused}_{i,x}$ is the relational abstraction of $\textsc{unused}_i$ (cf. Equation 3) in which we compare only the result stored in the variable x (i.e., we compare $\sigma[\omega](o)$ and $\sigma'[\omega](o)$, instead of $\sigma[\omega]$ and $\sigma'[\omega]$ as in Equation 3).

Theorem 5. *A terminating program does not use any of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{F}}$ of its non-interference abstraction Λ_{F} is a subset of $\mathcal{N}$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N} \Rightarrow P \models \mathcal{N}$$

[2] The case of a program using an input variable and then always diverging is not problematic because the analysis would be imprecise but still sound.

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N}$. By definition of γ_{F} (cf. Equation 17), since the program is terminating, we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$ and, by monotonicity of the concretization function $\gamma_{\rightsquigarrow}$ (cf. Equation 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N}$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}$. The conclusion follows from Theorem 4. □

Note that the termination of the program is necessary for the proof of Theorem 5. Indeed, for a non-terminating program, we have that $\Lambda_{\rightsquigarrow} \not\subseteq \gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$ (since $\Lambda_{\rightsquigarrow}$ includes relational abstractions of infinite traces that are missing from $\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$) and thus we cannot conclude the proof.

This result shows that the non-interference analysis Λ_{F} is an abstraction of the dependency semantics $\Lambda_{\rightsquigarrow}$ presented earlier. However, we remark that the same result applies to all other instances in this important class of analysis [5,25, etc.], which are therefore subsumed by our framework.

9 Strongly Live Variable Abstraction

Strongly live variable analysis [20] is a variant of the classic live variable analysis [32] performed by compilers to determine, for each program point, which variables may be potentially used before they are assigned to. A variable is *strongly live* if it is used in an assignment to another strongly live variable, or if is used in a statement other than an assignment. Otherwise, a variable is considered *faint*.

Strongly live variable analysis Λ_{X} is a *backward analysis* in the complete lattice $\langle \mathcal{P}(\mathrm{X}), \subseteq, \cup, \cap, \emptyset, \mathrm{X} \rangle$, where X is the set of all program variables. The transfer function $\Theta_{\mathrm{X}}[\![s]\!] \colon \mathcal{P}(\mathrm{X}) \to \mathcal{P}(\mathrm{X})$ for each statement s is defined as:

$$\Theta_{\mathrm{X}}[\![\texttt{skip}]\!](S) \stackrel{\text{def}}{=} S$$

$$\Theta_{\mathrm{X}}[\![x = e]\!](S) \stackrel{\text{def}}{=} \begin{cases} (S \setminus \{x\}) \cup \textsc{vars}(e) & x \in S \\ S & \text{otherwise} \end{cases}$$

$$\Theta_{\mathrm{X}}[\![\texttt{if } b\texttt{: } s_1 \texttt{ else: } s_2]\!](S) \stackrel{\text{def}}{=} \textsc{vars}(b) \cup \Theta_{\mathrm{X}}[\![s_1]\!](S) \cup \Theta_{\mathrm{X}}[\![s_2]\!](S)$$

$$\Theta_{\mathrm{X}}[\![\texttt{while } b\texttt{: } s]\!](S) \stackrel{\text{def}}{=} \textsc{vars}(b) \cup \Theta_{\mathrm{X}}[\![s]\!](S)$$

$$\Theta_{\mathrm{X}}[\![s_1\ s_2]\!](S) \stackrel{\text{def}}{=} \Theta_{\mathrm{X}}[\![s_1]\!] \circ \Theta_{\mathrm{X}}[\![s_2]\!](S)$$

where $\textsc{vars}(e)$ is the set of variables in the expression e. For input data usage, the initial set of strongly live variables contains the output variables of the program.

Example 9. Let us consider again the program P shown in Example 7. The strongly live variable analysis begins from the set $\{\texttt{passing}\}$ containing the output variable `passing`. At line 3, the set of strongly live variables is $\{\texttt{math}, \texttt{bonus}\}$ since `bonus` is used in an assignment to the strongly live variable `passing`, and `math` is used in the condition of the `if` statement. Finally, at line 1, the set of strongly live variables is $\{\texttt{english}, \texttt{math}, \texttt{bonus}\}$ because `english` is used in the condition of the `if` statement at line 2. Thus, strongly live variable analysis is able to conclude that the input variable `science` is unused. However, it is not precise enough to determine that the variable `english` is also unused. ■

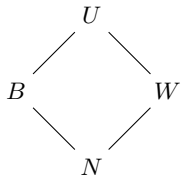

Fig. 5. Hasse diagram for the complete lattice $\langle \text{USAGE}, \sqsubseteq_{\text{USAGE}}, \sqcup_{\text{USAGE}}, \sqcap_{\text{USAGE}}, N, U \rangle$.

The imprecision of the analysis derives from the fact that it does not capture implicit flows of information precisely (cf. Section 8) but only over-approximates their presence. Thus, the analysis is unable to detect when a conditional statement, for instance, modifies only variables that have no impact on the outcome of a program; a situation likely to arise due to a programming error, as shown in the previous example. However, in virtue of this imprecise treatment of implicit flows, we can show that strongly live variable analysis is sound for input data usage, even for non-terminating programs.

We define the concretization function $\gamma_{\text{X}} \colon \mathcal{P}(X) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\bot))$ as:

$$\gamma_{\text{X}}(S) \stackrel{\text{def}}{=} \{R \in \Sigma \times \Sigma_\bot \mid \forall i \in \text{X} \setminus S \colon \text{UNUSED}_i(R)\} \tag{18}$$

where we abuse notation and use UNUSED_i (cf. Equation 3) to also denote its dependency abstraction (cf. Equation 10). We now show that strongly live variable analysis is sound for proving that a program does not use the faint variables.

Theorem 6. *A program does not use a subset J of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\text{X}}$ of its strongly live variable abstraction Λ_{X} is a subset of $\mathcal{N}_J$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\text{X}}(\Lambda_{\text{X}})) \subseteq \mathcal{N}_J \Rightarrow P \models \mathcal{N}_J$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\text{X}}(\Lambda_{\text{X}})) \subseteq \mathcal{N}_J$. By definition of γ_{X} (cf. Equation 18), we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\text{X}}(\Lambda_{\text{X}})$ and, by monotonicity of $\gamma_{\rightsquigarrow}$ (cf. Equation 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\text{X}}(\Lambda_{\text{X}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\text{X}}(\Lambda_{\text{X}})) \subseteq \mathcal{N}_J$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$. The conclusion follows from Theorem 4. □

This result shows that also strongly live variable analysis is subsumed by our framework as it is an abstraction of the dependency semantics $\Lambda_{\rightsquigarrow}$.

10 Syntactic Dependency Abstractions

In the following, we derive a more precise data usage analysis based on *syntactic* dependencies between program variables. For simplicity, the analysis does not take program termination into account, but we discuss possible solutions at the end of the section. Due to space limitations, we only provide a terse description of the abstraction and refer to [36] for further details.

In order to capture implicit dependencies from variables appearing in boolean conditions of conditional and while statements, we track when the value of a variable is used or modified in a statement based on the level of nesting of the statement in other statements. More formally, each program variable maps to a value in the complete lattice shown in Figure 5: the values U (*used*) and N (*not-used*) respectively denote that a variable may be used and is not used at the current nesting level; the values B (*below*) and W (*overwritten*) denote that a variable may be used at a lower nesting level, and the value W additionally indicates that the variable is modified at the current nesting level.

A variable is used (i.e., maps to U) if it is used in an assignment to another variable that is used in the current or a lower nesting level (i.e., a variable that maps to U or B). We define the operator $\textsc{assign}[\![x = e]\!]$ to compute the effect of an assignment on a map $m\colon \mathrm{X} \to \textsc{usage}$, where X is the set of all variables:

$$\textsc{assign}[\![x = e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} W & y = x \wedge y \notin \textsc{vars}(e) \wedge m(x) \in \{U, B\} \\ U & y \in \textsc{vars}(e) \wedge m(x) \in \{U, B\} \\ m(y) & \text{otherwise} \end{cases} \tag{19}$$

The assigned variable is overwritten (i.e., maps to W), unless it is used in e.

Another reason for a variable to be used is if it appears in the boolean condition e of a statement that uses another variable or modifies another used variable (i.e., there exists a variable x that maps to U or W):

$$\textsc{filter}[\![e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & y \in \textsc{vars}(e) \wedge \exists x \in \mathrm{X}\colon m(x) \in \{U, W\} \\ m(y) & \text{otherwise} \end{cases} \tag{20}$$

We maintain a *stack* of these maps that grows or shrinks based on the level of nesting of the currently analyzed statement. More formally, a stack is a tuple $\langle m_0, m_1, \ldots, m_k \rangle$ of mutable length k, where each element $m_0, m_1, \ldots, m_k$ is a map from X to $\textsc{usage}$. In the following, we use Q to denote the set of all stacks, and we abuse notation by writing $\textsc{assign}[\![x = e]\!]$ and $\textsc{filter}[\![e]\!]$ to also denote the corresponding operators on stacks:

$$\textsc{assign}[\![x = e]\!](\langle m_0, m_1, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \langle \textsc{assign}[\![x = e]\!](m_0), m_1, \ldots, m_k \rangle$$
$$\textsc{filter}[\![e]\!](\langle m_0, m_1, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \langle \textsc{filter}[\![e]\!](m_0), m_1, \ldots, m_k \rangle$$

The operator $\textsc{push}$ duplicates the map at the top of the stack and modifies the copy using the operator $\textsc{inc}$, to account for an increased nesting level:

$$\textsc{push}(\langle m_0, m_1, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \langle \textsc{inc}(m_0), m_0, m_1, \ldots, m_k \rangle$$
$$\textsc{inc}(m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} B & m(y) \in \{U\} \\ N & m(y) \in \{W\} \\ m(y) & \text{otherwise} \end{cases} \tag{21}$$

A used variable (i.e., mapping to U) becomes used below (i.e., now maps to B), and a modified variable (i.e., mapping to W) becomes unused (i.e., now maps

```
math, bonus ↦ U, passing ↦ W ⊔Q passing ↦ U = math, bonus, passing ↦ U
if not math:
    bonus ↦ U,  passing ↦ W | passing ↦ U
    passing = bonus
    passing ↦ B | passing ↦ U
passing ↦ U
```

Fig. 6. Data usage analysis of the last statement of the program shown in Example 7. Stack elements are separated by | and, for brevity, variables mapping to N are omitted.

to N). The dual operator POP combines the two maps at the top of the stack:

$$\begin{aligned} \text{POP}(\langle m_0, m_1, \dots, m_k \rangle) &\stackrel{\text{def}}{=} \langle \text{DEC}(m_0, m_1), \dots, m_k \rangle \\ \text{DEC}(m, k) &\stackrel{\text{def}}{=} \lambda y. \begin{cases} k(y) & m(y) \in \{B, N\} \\ m(y) & \text{otherwise} \end{cases} \end{aligned} \tag{22}$$

where the DEC operator restores the value a variable y mapped to before increasing the nesting level (i.e., $k(y)$) if it has not changed since (i.e., if the variable still maps to B or N), and otherwise retains the new value y maps to.

We can now define the data usage analysis Λ_{Q}, which is a *backward analysis* on the lattice $\langle \text{Q}, \sqsubseteq_{\text{Q}}, \sqcup_{\text{Q}} \rangle$. The partial order $\sqsubseteq_{\text{Q}}$ and the least upper bound $\sqcup_{\text{Q}}$ are the pointwise lifting, for each element of the stack, of the partial order and least upper bound between maps from X to USAGE (which in turn are the pointwise lifting of the partial order $\sqsubseteq_{\text{USAGE}}$ and least upper bound $\sqcup_{\text{USAGE}}$ of the USAGE lattice, cf. Figure 5). We define the transfer function $\Theta_{\text{Q}}[\![s]\!] \colon \text{Q} \to \text{Q}$ for each statement s in our simple programming language as follows:

$$\begin{aligned} \Theta_{\text{Q}}[\![\texttt{skip}]\!](q) &\stackrel{\text{def}}{=} q \\ \Theta_{\text{Q}}[\![x = e]\!](q) &\stackrel{\text{def}}{=} \text{ASSIGN}[\![x = e]\!](q) \\ \Theta_{\text{Q}}[\![\texttt{if } b\texttt{: } s_1 \texttt{ else: } s_2]\!](q) &\stackrel{\text{def}}{=} \text{POP} \circ \text{FILTER}[\![b]\!] \circ \Theta_{\text{Q}}[\![s_1]\!] \circ \text{PUSH}(q) \\ &\quad \sqcup_{\text{Q}} \text{POP} \circ \text{FILTER}[\![b]\!] \circ \Theta_{\text{Q}}[\![s_2]\!] \circ \text{PUSH}(q) \\ \Theta_{\text{Q}}[\![\texttt{while } b\texttt{: } s]\!](q) &\stackrel{\text{def}}{=} \text{lfp}_t^{\sqsubseteq_{\text{Q}}}\ \Theta_{\text{Q}}[\![\texttt{if } b\texttt{: } s \texttt{ else: skip}]\!] \\ \Theta_{\text{Q}}[\![s_1\ s_2]\!](q) &\stackrel{\text{def}}{=} \Theta_{\text{Q}}[\![s_1]\!] \circ \Theta_{\text{Q}}[\![s_2]\!](q) \end{aligned}$$

The initial stack contains a single map, in which the output variables map to the value U, and all other variables map to N. We exemplify the analysis below.

Example 10. Let us consider again the program P shown in Example 7. The initial stack begins with a single map m, in which the output variable `passing` maps to U and all other variables map to N.

At line 4, before analyzing the body of the conditional statement, a modified copy of m is pushed onto the stack: this copy maps `passing` to B, meaning that `passing` is only used in a lower nesting level, and all other variables still map to N (cf. Equation 21). As a result of the assignment (cf. Equation 19),

`passing` is overwritten (i.e., maps to W), and bonus is used (i.e., maps to U). Since the body of the conditional statement modifies a used variable and uses another variable, the analysis of its boolean condition makes `math` used as well (cf. Equation 20). Finally, the maps at the top of the stack are merged and the result maps `math`, `bonus`, and `passing` to U, and all other variables to N (cf. Equation 22). The analysis is visualized in Figure 6.

The stack remains unchanged at line 3 and line 2, since the statement at line 3 is identical to line 4 and the body of the conditional statement at line 2 does not modify any used variable and does not use any other variable. Finally, at line 1 the variable `passing` is modified (i.e., it now maps to W), while `math` and `bonus` remain used (i.e., they map to U). Thus, the analysis is precise enough to conclude that the input variables `english` and `science` are unused. ■

Note that, similarly to the non-interference analysis presented in Section 8, the data usage analysis Λ_{Q} does not consider non-termination. Indeed, for the program shown in Example 8, the analysis does not capture that the input variable `english` is used, even though the termination of the program depends on its value. We define the concretization function $\gamma_{\mathrm{Q}} \colon \mathrm{Q} \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$ as:

$$\gamma_{\mathrm{Q}}(\langle m_0, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \{R \in \Sigma \times \Sigma \mid \forall i \in \mathrm{X} \colon m_0(i) \in \{N\} \Rightarrow \textsc{unused}_i(R)\} \quad (23)$$

where again we write $\textsc{unused}_i$ (cf. Equation 3) to also denote its dependency abstraction. We now show that Λ_{Q} is sound for proving that a program does not use a subset of its input variables, *if the program is terminating*.

Theorem 7. *A terminating program does not use a subset J of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{Q}}$ of its abstraction Λ_{Q} is a subset of $\mathcal{N}_J$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J \Rightarrow P \models \mathcal{N}_J$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J$. Since the program is terminating, we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})$, by definition of the concretization function γ_{Q} (cf. Equation 23). Then, by monotonicity of $\gamma_{\rightsquigarrow}$ (cf. Equation 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$. The conclusion follows from Theorem 4. □

In order to take termination into account, one could map each variable appearing in the guard of a loop to the value U. Alternatively, one could run a termination analysis [3,33,34], along with the data usage analysis, and only map to U variables appearing in the loop guard of a possibly non-terminating loop.

11 Piecewise Abstractions

The static analyses presented so far can be used only to detect unused data stored in program variables. However, realistic data science applications read and manipulate data organized in data structures such as arrays, lists, and dictionaries. In the following, we demonstrate that having expressed the analyses

as abstract domains allows us to easily lift the analyses to such a scenario. In particular, to detect unused chunks of the input data, we combine the more precise data usage analysis presented in the previous section with the array content abstraction proposed by Cousot et al. [16]. Due to space limitations, we provide only an informal description of the resulting abstract domain and refer to [36] for further details and examples. The analyses presented in earlier sections can be similarly combined with the array abstraction for the same purpose.

We extend our small programming language introduced in Section 7 with integer variables, arithmetic and boolean comparison expressions, and arrays:

$$e ::= \cdots \mid a[e] \mid \texttt{len}(a) \mid e \ \oplus \ e \mid e \ \bowtie \ e \qquad \text{(expressions)}$$
$$s ::= \cdots \mid a[e] = e \qquad \text{(statements)}$$

where $\oplus$ and $\bowtie$ respectively range over arithmetic and boolean comparison operators, a ranges over array variables, and $\texttt{len}(a)$ denotes the length of a.

Piecewise Array Abstraction. The array abstraction [16] divides an array into consecutive segments, each segment being a uniform abstraction of the array content in that segment. The bounds of the segments are specified by sets of side-effect free expressions restricted to a canonical normal form, all having the same (concrete) value. The abstraction is parametric in the choice of the abstract domains used to manipulate sets of expressions and to represent the array content within each segment. For our analysis, we use the octagon abstract domain [31] for the expressions, and the USAGE lattice presented in the previous section (cf. Figure 5) for the segments. Thus, an array a is abstracted, for instance, as $\{0, i\}\ N\ \{j+1\}?\ U\ \{\texttt{len}(a)\}$, where the symbol ? indicates that the segment $\{0, i\}\ N\ \{j+1\}$ might be empty. The abstraction indicates that all array elements (if any) from index i (which is equal to zero) to index j (the bound $j+1$ is exclusive) are unused, and all elements from $j+1$ to $\texttt{len}(a)-1$ may be used. Let A be the set of all such array abstractions. The initial segmentation of an array $a \in \mathrm{A}$ is a single segment with unused content (i.e., $\{0\}\ N\ \{\texttt{len}(a)\}?$).

For our analysis, we augment the array abstraction with new backward assignment and filter operators. The operators $\textsc{assign}_{\mathrm{A}}[\![a[i] = e]\!]$ and $\textsc{filter}_{\mathrm{A}}[\![e]\!]$ split and fill segments to take into account assignments and accesses to array elements that influence the program outcome. For instance, an assignment to $a[i]$ with an expression containing a used variable modifies the segmentation $\{0\}\ N\ \{\texttt{len}(a)\}?$ into $\{0\}\ N\ \{i\}?\ U\ \{i+1\}\ N\ \{\texttt{len}(a)\}?$, which indicates that the array element at index i is used by the program. An access $a[i]$ in a boolean condition guarding a statement that uses or modifies another used variables is handled analogously. Instead, the operator $\textsc{assign}_{\mathrm{A}}[\![x = e]\!]$ modifies the segmentation of an array by replacing each occurrence of the assigned variable x with the canonical normal form of the expression e. For instance, an assignment $i = i+1$ modifies the segmentation $\{0\}\ N\ \{i\}?\ U\ \{i+1\}\ N\ \{\texttt{len}(a)\}?$ into $\{0\}\ N\ \{i+1\}?\ U\ \{i+2\}\ N\ \{\texttt{len}(a)\}?$. If e cannot be precisely put into a canonical normal form, the operator replaces the assigned variable with an approximation of e as an integer interval [13] computed using the underlying numerical

```
1 failed = 0
2 i = 1                                                    # 1 should be 0
3 while i < len(grades):
4     if grades[i] < 4: failed = failed + 1
5     i = i + 1
6 passing = 2 * failed < len(grades)
```

Fig. 7. Another program to check if a student has passed a number of exams based on their grades stored in the array `grades`. The programmer has made a mistake at line 2 that causes the program to ignore the grade stored at index 0 in `grades`.

$\texttt{grades} \mapsto \{0\}\, N\, \{\texttt{i}\}?\, U\, \{\texttt{i}+1\}?\, U\, \{\texttt{len(grades)}\}?$
`while` $\texttt{i} < \texttt{len(grades)}$:
    $\texttt{grades} \mapsto \{0\}\, N\, \{\texttt{i}\}?\, U\, \{\texttt{i}+1\}?\, B\, \{\texttt{i}+2\}?\, B\, \{\texttt{len(grades)}\}? \mid \ldots$
    `if` $\texttt{grades[i]} < 4$:
        $\texttt{grades} \mapsto \{0\}\, N\, \{\texttt{i}+1\}?\, B\, \{\texttt{i}+2\}?\, B\, \{\texttt{len(grades)}\}? \mid \cdots \mid \ldots$
        $\texttt{failed} = \texttt{failed} + 1$
        $\texttt{grades} \mapsto \{0\}\, N\, \{\texttt{i}+1\}?\, B\, \{\texttt{i}+2\}?\, B\, \{\texttt{len(grades)}\}? \mid \cdots \mid \ldots$
    $\texttt{grades} \mapsto \{0\}\, N\, \{\texttt{i}+1\}?\, B\, \{\texttt{i}+2\}?\, B\, \{\texttt{len(grades)}\}? \mid \ldots$
    $\texttt{i} = \texttt{i} + 1$
    $\texttt{grades} \mapsto \{0\}\, N\, \{\texttt{i}\}?\, B\, \{\texttt{i}+1\}?\, B\, \{\texttt{len(grades)}\}? \mid \ldots$
$\texttt{grades} \mapsto \{0\}\, N\, \{\texttt{len(grades)}\}?$

Fig. 8. Data usage analysis of the loop statement of the program shown in Example 11. Stack elements are separated by | and, for brevity, only array variables are shown.

domain, and possibly merges segments together as a result of the approximation. For instance, a non-linear assignment $i = i * j$ approximated as $i = [0, 1]$ modifies the segmentation $\{0\}\, N\, \{i\}?\, U\, \{i+1\}\, N\, \{\texttt{len}(a)\}?$ into $\{0\}\, U\, \{2\}\, N\, \{\texttt{len}(a)\}?$, which loses the information that the initial segment of the array is unused.

When merging control flows, segmentations are compared or joined by means of a *unification algorithm* [16], which finds the coarsest common refinement of both segmentations. Then, the comparison $\sqsubseteq_{\mathrm{A}}$ or the join $\sqcup_{\mathrm{A}}$ is performed pointwise for each segment using the corresponding operators of the underlying abstract domain chosen to abstract the array content. For our analysis, we adapt and refine the originally proposed unification algorithm to take into account the knowledge of the numerical domain chosen to abstract the segment bounds. We refer to [36] for further details. A widening ∇_{A} limits the number of segments to enforce termination of the analysis.

Piecewise Data Usage Analysis. We can now map each scalar variable to an element of the USAGE lattice and each array variable to an array segmentation in A, and use the data usage analysis Λ_{Q} presented in the previous section to identify unused input data stored in variables and portions of arrays.

Example 11. Let us consider the program shown in Figure 7 where the array variable `grades` and the variable `passing` are the input and output variables, respectively. The initial stack contains a single map in which `passing` maps to U,

all other scalar variables map to N, and `grades` maps to $\{0\}\, N\, \{\texttt{len(grades)}\}?$, indicating that all elements of the array (if any) are unused.

At line 6, the assignment modifies the variable `passing` (i.e., `passing` now maps to W) and uses the variable `failed` (i.e., `failed` now maps to U), while every other variable remains unchanged.

The result of the analysis of the loop statement at line 3 is shown in Figure 8. The analysis of the loop begins by pushing (cf. Equation 21) a map onto the stack in which `passing` becomes unused (i.e., maps to N) and `failed` is used only in a lower nesting level (i.e., maps to B), and every other variable still remains unchanged. At the first iteration of the analysis of the loop body, the assignment at line 4 uses `failed` and thus the access `grades[i]` at line 3 creates a used segment in the segmentation for `grades`, which becomes $\{0\}\, N\, \{\texttt{i}\}?\, U\, \{\texttt{i}+1\}\ \ N\, \{\texttt{len(grades)}\}?$. At the second iteration, the PUSH operator turns the used segment $\{\texttt{i}\}\, U\, \{\texttt{i}+1\}$ into $\{\texttt{i}\}\, B\, \{\texttt{i}+1\}$, and the assignment to `i` modifies the segment into $\{\texttt{i}+1\}\, B\, \{\texttt{i}+2\}$ (while the segmentation in the second stack element becomes $\{0\}\, N\, \{\texttt{i}+1\}?\, U\, \{\texttt{i}+2\}\ \ N\, \{\texttt{len(grades)}\}?$). Then, the access to the array at line 3 creates again a used segment $\{\texttt{i}\}\, U\, \{\texttt{i}+1\}$ (in the first segmentation) and the analysis continues with the result of the POP operator (cf. Equation 22): $\{0\}\, N\, \{\texttt{i}\}?\, U\, \{\texttt{i}+1\}?\, U\, \{\texttt{i}+2\}?\, N\, \{\texttt{len(grades)}\}?$. After widening, the last two segments are merged into a single segment, and the analysis of the loop terminates with $\{0\}\, N\, \{\texttt{i}\}?\, U\, \{\texttt{i}+1\}?\, U\, \{\texttt{len(grades)}\}?$.

Finally, the analysis of the assignment at line 2 produces the segmentation $\{0\}\, N\, \{1\}?\ U\, \{2\}?\ U\, \{\texttt{len(grades)}\}?$, which correctly indicates that the first element of the array `grades` (if any) is unused by the program. ■

Implementation. The analyses presented in this and in the previous section are implemented in the prototype static analyzer LYRA and are available online[3].

The implementation is in PYTHON and, at the time of writing, accepts programs written in a limited subset of PYTHON without user-defined classes. A type inference is run before the analysis of a program. The analysis is performed backwards on the control flow graph of the program with a standard worklist algorithm [32], using widening at loop heads to enforce termination.

12 Related Work

The most directly relevant work has been discussed throughout the paper. The non-interference analysis proposed by Assaf et al. [6] (cf. Section 8) is similar to the logic of Amtoft and Banerjee [5] and the type system of Hunt and Sands [25]. The data usage analysis proposed in Section 10 is similar to dependency analyses used for program slicing [37] (e.g., [24]). Both analyses as well as strongly live variable analysis (cf. Section 9) are based on the *syntactic* presence of a variable in the definition of another variable. To overcome this limitation, one should look further for *semantic* dependencies between *values* of program variables. In

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

this direction, Giacobazzi, Mastroeni, and others [19,22,29] have proposed the notion of *abstract dependency*. However, note that an analysis based on abstract dependencies would over-approximate the subset of the input variables that are unused by a program. Indeed, the absence of an abstract dependency between variables (e.g., a dependency between the parity of the variables [19,29]) does not imply the absence of a (concrete) dependency between the variables (i.e., a dependency between the values of the variables). Thus, such an analysis could not be used to prove that a program *does not use* a subset of its input variables, but would be used to prove that a program *uses* a subset of its input variables.

Semantics formulations using *sets of sets of traces* have already been proposed in the literature [6,28]. Mastroeni and Pasqua [28] lift the hierarchy of semantics developed by Cousot [12] to sets of sets of traces to obtain a hierarchy of semantics suitable for verifying general program properties (i.e., properties that are not subset-closed, cf. Section 7). However, *none* of the semantics that they proposed is suitable for input data usage: all semantics in the hierarchy are abstractions of a semantics that contains sets with both finite and infinite traces and thus, unlike our outcome semantics (cf. Section 5), cannot be used to reason about terminating and non-terminating outcomes of a program. Similarly, as observed in [28], the semantics proposed by Assaf et al. [6] can be used to verify only subset-closed properties. Thus, it cannot be used for input data usage.

Finally, to the best of our knowledge, our work is the first to aim at detecting programming errors in data science code using static analysis. Closely related are [7,10] which, however, focus on spreadsheet applications and target errors in the data rather than the code that analyzes it. Recent work [2] proposes an approach to repair *bias* in data science code. We believe that our work can be applied in this context to prove absence of bias, e.g., by showing that a program does not use gender information to decide whether to hire a person.

13 Conclusion and Future Work

In this paper, we have proposed an abstract interpretation framework to automatically detect input data that remains unused by a program. Additionally, we have shown that existing static analyses based on dependencies are subsumed by our unifying framework and can be used, with varying degrees of precision, for proving that a program does not use some of its input data. Finally, we have proposed a data usage analysis for more realistic data science applications that store input data in compound data structures such as arrays or lists.

As part of our future work, we plan to use our framework to guide the design of new, more precise static analyses for data usage. We also want to explore the complementary direction of proving that a program *uses* its input data by developing an analysis based on abstract dependencies [19,22,29] between program variables, as discussed above. Additionally, we plan to investigate other applications of our work such as provenance or lineage analysis [9] as well as proving absence of algorithmic bias [2]. Finally, we want to study other programming errors related to data usage such as accidental data duplication.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A Core Calculus of Dependency. In *POPL*, pages 147–160, 1999.
2. A. Albarghouthi, L. D'Antoni, and S. Drews. Repairing Decision-Making Programs Under Uncertainty. In *CAV*, pages 181–200, 2017.
3. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS*, pages 117–133, 2010.
4. B. Alpern and F. B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, 1985.
5. T. Amtoft and A. Banerjee. Information Flow Analysis in Logical Form. In *SAS*, pages 100–115, 2004.
6. M. Assaf, D. A. Naumann, J. Signoles, E. Totel, and F. Tronel. Hypercollecting Semantics and Its Application to Static Analysis of Information Flow. In *POPL*, pages 874–887, 2017.
7. D. W. Barowy, D. Gochev, and E. D. Berger. CheckCell: Data Debugging for Spreadsheets. In *OOPSLA*, pages 507–523, 2014.
8. D. Binkley and K. B. Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.
9. J. Cheney, A. Ahmed, and U. A. Acar. Provenance as Dependency Analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.
10. T. Cheng and X. Rival. Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection. In *ESOP*, 2015.
11. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
12. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
13. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Symposium on Programming*, pages 106–130, 1976.
14. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
15. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL*, pages 269–282, 1979.
16. P. Cousot, R. Cousot, and F. Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *POPL*, pages 105–118, 2011.
17. D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.
18. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.
19. R. Giacobazzi and I. Mastroeni. Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation. In *POPL*, pages 186–197, 2004.
20. R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of Approximate Semantics with Respect to Program Transformations. In *GI - 11. Jahrestagung*, pages 1–10, 1981.
21. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *S & P*, pages 11–20, 1982.

22. R. Halder and A. Cortesi. Abstract Program Slicing on Dependence Condition Graphs. *Science of Computer Programming*, 78(9):1240–1263, 2013.
23. T. Herndon, M. Ash, and R. Pollin. Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff. *Cambridge Journal of Economics*, 38(2):257–279, 2014.
24. S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
25. S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *POPL*, pages 79–90, 2006.
26. L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
27. N. G. Leveson and C. S. Turner. Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.
28. I. Mastroeni and M. Pasqua. Hyperhierarchy of Semantics. In *SAS*, 2017.
29. I. Mastroeni and D. Zanardini. Abstract Program Slicing: An Abstract Interpretation-Based Approach to Program Slicing. *ACM Transactions on Computational Logic*, 18(1):7:1–7:58, 2017.
30. J. Mencinger, A. Aristovnik, and M. Verbic. The Impact of Growing Public Debt on Economic Growth in the European Union. *Amfiteatru Economic*, 16(35):403–414, 2014.
31. A. Miné. The Octagon Abstract Domain. *Higher Order and Symbolic Computation*, 19(1):31–100, 2006.
32. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
33. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.
34. C. Urban. The Abstract Domain of Segmented Ranking Functions. In *SAS*, pages 43–62, 2013.
35. D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
36. S. Wehrli. Static Program Analysis of Data Usage Properties. Master's thesis, ETH Zurich, Zurich, Switzerland, 2017.
37. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.