Abstract Semantic Dependency

Patrick Cousot
Courant Institute of Mathematical Sciences, New York University

Abstract. Dependency is a prevalent notion in computer science. There have been numerous informal or formal attempts to define viable syntactic and semantic concepts of dependency in programming languages with subtle variations and limitations. We develop a new value dependency analysis defined by abstract interpretation of a trace semantics. A sound approximate dependency algorithm is formally derived by calculational design. Further abstractions provide information flow, slicing, non-interference, dye, and taint analyses.

1 Introduction

Motivation: Dependency is a prevalent notion in computer science. For example it is useful in program development [14], it is an important part of any parallelizing compiler [54]. It appears in dataflow analysis [51,64], (abstract) program slicing [68,56,49,5], program refactoring [7], hardware design [2] and debugging [46]. It is prevailing in security, [30,12] including privacy analysis [59,47], and in data bases [27].

Context: There have been numerous attempts to define a viable semantic concept of dependency in programming languages. For example they are purely syntactic, not taking data into account [68]. Or they are postulated on programs [26,6,1] or on one execution trace [69] rather than derived from a definition of dependency and a definition of the program semantics. Or they are limited to one [17] or a few [48] of the many possible definitions of dependency based on a specific instrumentation of the semantics of a given language. Or they make assumptions of when dependencies are observed *e.g.* on program termination only [30,8,11] maybe including nontermination [5,64]. These are typical limitations that we would like to overcome.

Objective: Our aim is to introduce, justify, and illustrate a methodology to define flexible concepts of dependency and corresponding static analyzes that can be adapted to various contexts of use, each context requiring different notions of dependency, sometimes with subtle variations.

The general idea of dependency is that modifying something in an execution will later modify some other thing in the execution. This involves comparing at least two executions, the original and the modified one. Therefore dependency is not a property of a trace (such as invariance and termination) but a property of a set of traces (such as program equivalence), sometimes called hyperproperty [18].

Previous definitions of dependency (and related notions such as interference) have called for changing the description of program executions by considering multisemantics [16] or multilogics [31] handling more than one execution at a time. Other abstract interpretation-based definitions of dependency consider

B.-Y. E. Chang (Ed.): SAS 2019, LNCS 11822, pp. 1–22, 2019.
https://doi.org/10.1007/978-3-030-32304-2_6

only one execution trace (by postulating dependency on that execution trace [69] or by annotating the semantics [17,28]). When considering several execution traces, dependency can be defined by abstracting to functional dependency [56]. Otherwise, one can provide an hypercollecting semantics [8,64] which then abstracted.

As usual in abstract interpretation [23], we represent properties of entities in a universe $\mathbb{U}$ by a subset of this universe. So a property of elements of $\mathbb{U}$ belongs to $\wp(\mathbb{U})$. For example "to be a natural" is the property $\mathbb{N} \triangleq \{n \in \mathbb{Z} \mid n \geqslant 0\}$ of the integers $\mathbb{Z}$. The property "n is a natural" is "$n \in \mathbb{N}$".

Given a program component $\mathtt{s}$ which semantics $\mathscr{S}[\![\mathtt{s}]\!]$ is an element of the semantic domain $\mathscr{D}[\![\mathtt{s}]\!]$, we understand a program component property P as a property of its semantics $\mathscr{S}[\![\mathtt{s}]\!] \in \mathscr{D}[\![\mathtt{s}]\!]$ so $P \in \wp(\mathscr{D}[\![\mathtt{s}]\!])$ and $\mathscr{S}[\![\mathtt{s}]\!] \in P$ means that $\mathscr{S}[\![\mathtt{s}]\!]$ has property P. The *collecting semantics* is the strongest program property, that is the singleton $\{\mathscr{S}[\![\mathtt{s}]\!]\}$.

For example, the semantics we consider is a relation between a finite execution trace representing a past computation into its continuation into the future which may not terminate so $\mathscr{D}[\![\mathtt{s}]\!] = \wp(\mathbb{T}^+ \times \mathbb{T}^{+\infty})$ where $\mathbb{T}^+$ is the set of all finite execution traces and $\mathbb{T}^{+\infty}$ the set of all finite or infinite execution traces. So program properties belong to $\wp(\wp(\mathbb{T}^+ \times \mathbb{T}^{+\infty}))$. They are often called "hyper properties", after [18]. This terminology is supposed to rectify a previous misunderstanding of program properties in [3], where property stands for a trace property. More precisely, a program semantics is a set of execution traces in $\wp(\mathbb{T}^{+\infty})$ and a program property is also a set of execution traces in $\wp(\mathbb{T}^{+\infty})$. So a semantics and its properties belong to the same semantic domain, which is apparently incoherent.

Considering a property as a set of entities (with this property) has several advantages. It applies to languages which semantics are not naturally defined as traces *e.g.* [51] for logic programs. It avoids the definition of program properties through program transformation (like [10] duplicating programs which can compare one execution to another one but not one too many other ones). It eliminates the expressivity problems of logics (which can always be taken into account by a further abstraction). It eliminates the need to define different notions of properties for different notions of entities. In particular, the abstraction of a property is a property such as $\langle \wp(\wp(\mathbb{T}^{+\infty})), \subseteq \rangle \underset{\alpha_\cup}{\overset{\gamma_\cup}{\leftrightarrows}} \langle \wp(\mathbb{T}^{+\infty}), \subseteq \rangle$ with $\alpha_\cup(X) \triangleq \cup X$, thus solving the apparent incoherence of [3]. Finally, and more importantly, it aims at avoiding to create different theories for concepts that are the same.

One difficulty encountered *e.g.* by [8,64] to define dependency is to lift the structural trace semantics of a program component in $\wp(\mathbb{T}^+ \times \mathbb{T}^{+\infty})$ into a structural collecting semantics in $\wp(\wp(\mathbb{T}^+ \times \mathbb{T}^{+\infty}))$. For example, [8] has $\mathscr{D}[\![\mathtt{s}]\!] = \mathbf{Trc}_\perp \to \mathbf{Trc}_\perp$ (where $\mathbf{Trc}$ is a set of pairs of initial-final states augmented by $\perp$ for non-termination) while the (hyper-)collecting semantics is in $\wp(\wp(\mathbf{Trc})) \to \wp(\wp(\mathbf{Trc}))$ not in $\wp(\mathbf{Trc}_\perp \to \mathbf{Trc}_\perp)$. This is a strict approximation (as shown by [8, Theorem 1] which is an inclusion not an equality). Similarly, [64] uses an "outcome semantics" which approximates the (hyper-)collecting semantics.

These collecting semantics are specialized for dependency and lack generality since traces are approximated by a relation or function, but the advantage is that dependency boils down to functional dependency, which is easy to define [56]. We show that, for dependency, we can dispense with the formal structural definition of the (hyper-)collecting semantics $\{\mathcal{S}[\![\mathtt{S}]\!]\}$ (since it is trivially isomorphic to $\mathcal{S}[\![\mathtt{S}]\!]$ by the singleton map $\bullet \rightarrow \{\bullet\}$).

Content: We consider the syntax and trace semantics of iterative programs as defined in [20, Section **2**] in this volume. Traces are necessary to allow us to observe sequences of values of variables, in particular infinite ones. More abstract input/output semantics (such as denotational, natural, or axiomatic semantics) would not be adequate since intermediate or infinite computations are abstracted away. Informal requirements on the semantic definition of dependency are illustrated in Section **3**. The formal definition of value dependency is in Section **4**. We prove that this definition is valid both for prefix and maximal trace semantics hence excludes timing channels including empty observations. The calculational design of the structural static potential value dependency analysis is in Section **5**. It is not postulated without justification but designed by abstract interpretation of the semantics, handling uniformly the control and data dependency notions of [26]. Dye and tracking analysis are further abstractions described in Section **7**. We discuss related work and conclude in Section **2**.

2 Syntax and Trace Semantics

We consider a subset of `C` with simple variables, arithmetic and boolean expressions, assignment, skip (`;`), conditionals, `while` iterations, break, compound statement, statement lists. The syntax, program labelling, prefix trace semantics, and maximal trace semantics of this subset of `C` is defined in [20, Section **2**] in this volume. The main idea is that $\langle \pi_0, \pi_1 \rangle \in \mathcal{S}[\![\mathtt{S}]\!]$ if and only if the trace π_0 representing a past computation arriving at `S` is continued within `S` by π_1 resulting in a computation $\pi_0 \frown \pi_1$. The continuation trace π_1 is finite prefix of the whole computation when $\mathcal{S}[\![\mathtt{S}]\!] = \mathcal{S}^*[\![\mathtt{S}]\!]$. π_1 is finite maximal or infinite when $\mathcal{S}[\![\mathtt{S}]\!] = \mathcal{S}^{+\infty}[\![\mathtt{S}]\!]$. $\varrho(\pi)\mathtt{x}$ denotes the value of a variable `x` at the end of the trace π.

3 Informal Requirements for a Semantic Definition of Dependency

According to [26], "Information *flows* from object `x` to object `y`, denoted `x` $\rightsquigarrow$ `y`, whenever information stored in `x` is transferred to, or used to derive information transferred to, object `y`". When `x` $\rightsquigarrow$ `y` we say that `x` *flows* to `y` or, when considering the inverse relation, that `y` *depends* on `x`. To make this information flow clear, most definitions of in/dependency [17,65,64]) are of the form "changing (part of) the input (say `x`) may/should not change (part of) the output (say `y`)", sometimes including nontermination [64]. For example, a pure function depends

on a parameter if and only if changing only this parameter changes the result of the function. In non-interference [30,48] changing private/untrusted input data should not change public/trusted output data. This shows that two different executions reflecting the change should be involved in the definition of dependency (or secrecy in [65]).

Dependency is usually *static* (valid for any execution of the program). The dependency relation $\rightsquigarrow$ can be *global* (valid anywhere in the program as in [26]) or *local* (that is relative to a correspondence between initial values of variables and their values when reaching a program point, if ever, including for nonterminating executions). We consider a *static* and *local* definition of dependency. The following examples illustrate this intuition and show how it may be made more precise.

Example 1 (explicit dependency). Consider ℓ_1 `y = x ;` ℓ_2. Changing the initial value y_0 of `y` will change the value of `y` at the entry point ℓ_1. Changing the initial value x_0 of `x` will not change the value of `y` at ℓ_1. So at the entry point ℓ_1, `y` depends on y_0 but not on x_0.

The value of `y` at exit point ℓ_2 is x_0 so changing the initial value y_0 of `y` will not change the value of `y` at ℓ_2. Changing the initial value x_0 of `x` will change the value of `y` at ℓ_2. So at ℓ_2, `y` depends on x_0 but not on y_0. Such a dependency at ℓ_2 is called explicit in [26] since it does not depend on the program control.

Dependency is local since `x` $\not\rightsquigarrow$ `y` at ℓ_1 but `x` $\rightsquigarrow$ `y` at ℓ_2. We write `x` $\not\rightsquigarrow^{\ell_2}$ `y` and `x` $\rightsquigarrow^{\ell_2}$ `y` to show the program point where dependency is specified □

Example 2 (implicit dependency). Consider $\mathtt{P_a} \triangleq \ell_1$ `y = 1 ;if` ℓ_2 `(x == 0) { ;`ℓ_4 `}`ℓ_5. Changing the initial value x_0 of `x` will change whether program control ℓ_4 is reached or not. If ℓ_4 is reached then the value of `y` at ℓ_4 will always be 1 so `y` does not depend on x_0 at ℓ_4 (and neither at ℓ_5).

Consider now $\mathtt{P_b} \triangleq \ell_1$ `y = 1 ;if` ℓ_2 `(x == 0) {` ℓ_3 `y = x ;`ℓ_4 `}`ℓ_5. Changing the initial value x_0 of `x` will change whether program control points ℓ_3 and ℓ_4 are reached or not. If ℓ_4 is reached then the value of `y` at ℓ_4 will always be 0 so `y` does not depend on x_0 at ℓ_4. However, depending on the initial value x_0 of `x`, the value of `y` at ℓ_5 will be either 1 (when $x_0 \neq 0$) or 0 (when $x_0 = 0$) so `y` depends on x_0 at ℓ_5. Such a dependency at ℓ_5 is called implicit in [26] since it depends on the program control. □

Our formalization of dependency does not need to distinguish implicit dependency (Ex. 2) from explicit dependency (Ex. 1) since the definition is the same in both cases.

Example 3 (timely dependency). Consider the program `while (0 == 0)` ℓ `y = x ;`. The sequence of values taken by `x` at ℓ is x_0, x_0, x_0, …while it is y_0, x_0, x_0, …for `y`. So `x` depends on x_0 while `y` depends on x_0 and y_0 at ℓ. For `y` considering only one possible value during the iterations would be insufficient to determine the dependency upon initial values and, in general, we have to consider the full sequence of successive values of `y` at a given program point ℓ. □

Example 4 (value dependency). Consider the program `while (0 == 0) { ℓ x = x - 1 ;if (x == 0) y = y + 1 ; }`. If $x_0 \leqslant 0$, the sequence of values of `y` at ℓ is the infinite sequence y_0, y_0, y_0, …. If $x_0 > 0$, it is $\overbrace{y_0, \ldots, y_0}^{x_0\texttt{times}}$, $y_0 + 1$, $y_0 + 1$, …. So we can find two executions of the program with different initial values x_0 of `x` such that the sequences of values of `y` at ℓ have a common prefix but differ at least by one value after that prefix. So `y` depends on x_0 at ℓ. □

Example 5 (timing channel). Consider the program `int x, y; while (x > 0) ℓ x = x - 1 ;` (where we have added a declaration to show that the program involves variable `y`). If the initial value of `x` is $x_0 \leqslant 0$ then the sequence of values taken by `y` at ℓ is empty. Otherwise, if $x_0 > 0$, it is $y_0 \cdot y_0 \cdot \ldots \cdot y_0$ repeated x_0 times. So changing x_0 changes this sequence of values. Depending on x_0 we can find sequences of values of at ℓ that differ in length, but along these sequences we cannot find a point where they differ in value.

In security, this is a covert channel [44] (more precisely a timing channel [58]) which may or may not be considered as observable, the choice being application-dependent.

Traditionally, in dependency analysis, timing channels are not considered to be at the origin of dependencies [26], in particular when dependency is used in the context of compilation. □

Example 6 (empty observation). Consider the program `if (x==0) { ℓ₁ y = x ;ℓ₂ }ℓ₃`. What are the values of `y` observed at ℓ_2? If `x==0` this is 0 while if `x!=0` there is no possible observation `y` at ℓ_2. So we may consider that an empty observation is a valid observation in which case `y` depends on `x` at ℓ_2. This is certainly a frequent point of view in security. On the contrary, we may exclude empty observations, in which case `y` does not depend on `x` at ℓ_2. This is more common in compilation (since `y` is constant at ℓ_2). Notice that in both cases `y` depends on `x` at ℓ_3 since we can observe different values of `y` depending on the test on `x`. □

Our definition of dependency relies on the timely observation of values but excludes timing channels and empty observations. This is an arbitrary choice that follows the implicit tradition in compilation.

4 Formal semantic definition of value dependency

Informally, we say that the initial value x_0 of variable `x` flows to variable `y` at program point ℓ (or `y` depends on the initial value of `x`), written $\texttt{x} \rightsquigarrow^{\ell} \texttt{y}$, if and only if changing the initial value x_0 of `x` will change the sequence of values taken by `y` whenever execution reaches program point ℓ.

Sequence of values of a variable at a program point Given an initialization trace $\pi_0 \in \mathbb{T}^+$ followed by a nonempty trace $\pi \in \mathbb{T}^{+\infty}$, let us define the sequence $\mathsf{seqval}[\![\texttt{y}]\!]^{\ell}(\pi_0, \pi)$ of values of the variable `y` at program point ℓ along the trace π continuing π_0 as follows.

$$\mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0, \ell) \triangleq \boldsymbol{\varrho}(\pi_0)\mathsf{y} \tag{1}$$
$$\mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0, \ell') \triangleq \mathfrak{e} \qquad \text{when } \ell' \neq \ell$$
$$\mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0, \ell \xrightarrow{a} \ell''\pi) \triangleq \boldsymbol{\varrho}(\pi_0)\mathsf{y} \cdot \mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0 \frown \ell \xrightarrow{a} \ell'', \ell''\pi)$$
$$\mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0, \ell' \xrightarrow{a} \ell''\pi) \triangleq \mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0 \frown \ell' \xrightarrow{a} \ell'', \ell''\pi) \qquad \text{when } \ell' \neq \ell$$

$\mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0, \pi)$ is the empty sequence $\mathfrak{e}$ when ℓ does not appear in π. We rely on intuition that this definition applies to finite and infinite traces (by passing to the limit as in [20, Section **2.5**]).

The sequence of values of variable y at a program point ℓ abstracts away the position in traces where the values are observed. So execution time (represented as the number of steps that have been executed to reach a given position in the trace) is abstracted away.

Differences between sequences of values of a variable at a program point The definition of dependency of y on the initial value of x involves the comparison of sequences ω and ω' of the successive values of variable y taken at some program point for two different executions that differ on the initial value of x. By "differ", we mean that the sequences may have a common prefix but must eventually have a different value at some position in the sequences.

$$\mathsf{diff}(\omega, \omega') \triangleq \exists \omega_0, \omega_1, \omega'_1, \nu, \nu' \,.\, \omega = \omega_0 \cdot \nu \cdot \omega_1 \wedge \omega' = \omega_0 \cdot \nu' \cdot \omega'_1 \wedge \nu \neq \nu' \tag{2}$$

Observe that $\neg\mathsf{diff}(\omega, \omega')$ implies either that $\omega = \omega'$ (the futures are the same so there is no dependency) or one is a strict prefix of the other (this is a timing channel abstracted away in this definition (2) of dependency). Because ω and ω' in (2) must contain at least one value, they cannot be empty (thus excluding Ex. 6).

Definition of value dependency Let us define $\mathcal{D}^\ell\langle \mathsf{x}, \mathsf{y}\rangle$ to mean that "the sequence of values of variable y at ℓ depends upon the initial value of x", also written $\mathsf{x} \overset{\ell}{\rightsquigarrow} \mathsf{y}$ to mean that the initial value of x flows to y. So there are two execution traces whose initial values are the same but for x for which the sequences of values of y at program point ℓ on these two execution traces do differ.

Definition 1 (Dependency $\mathcal{D}$).

$$\begin{aligned}\mathcal{D}^\ell\langle \mathsf{x}, \mathsf{y}\rangle \triangleq \{\Pi \in \wp(\mathbb{T}^+ \times \mathbb{T}^{+\infty}) \mid \exists \langle \pi_0, \pi_1\rangle, \langle \pi'_0, \pi'_1\rangle \in \Pi \,.\, \\ (\forall \mathsf{z} \in \mathbb{V} \setminus \{\mathsf{x}\} \,.\, \boldsymbol{\varrho}(\pi_0)\mathsf{z} = \boldsymbol{\varrho}(\pi'_0)\mathsf{z}) \wedge \\ \mathsf{diff}(\mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi_0, \pi_1), \mathsf{seqval}[\![\mathsf{y}]\!]^\ell(\pi'_0, \pi'_1))\}\end{aligned} \tag{3}$$

□

We do not need to require $\boldsymbol{\varrho}(\pi_0)\mathsf{x} \neq \boldsymbol{\varrho}(\pi'_0)\mathsf{x}$ in (3), since, the language being deterministic, the computations would be the same for the same initial values of variables.

Dependency in (3) defines (a) an abstraction of the past (the initial value of variables), (b) an abstraction of the future ($\mathsf{seqval}[\![\mathsf{y}]\!]$), (c) the difference between past abstractions (the initial values of variables only differ for x), (d) the difference between futures (diff). It states that the abstraction of the future depends on the abstraction of the past if and only if there exist two executions with different past abstractions and different future abstractions.

Definition 2 (Value dependency flow). *At program point ℓ of program P, variable y depends on the initial value of variable x (or $\mathsf{x} \leadsto^{\ell}_{\mathsf{P}} \mathsf{y}$ i.e. the initial value of variable x flows to variable y at program point ℓ) if and only if*

$$\mathsf{x} \leadsto^{\ell}_{\mathsf{P}} \mathsf{y} \triangleq \big(\mathcal{S}^{+\infty}[\![\mathsf{P}]\!] \in \mathcal{D}^{\ell}\langle \mathsf{x}, \mathsf{y}\rangle\big). \tag{4}$$ □

The definition of seqval in (1) accounts for timely dependency (Ex. 3) while that of diff in (2) accounts for value dependency (Ex. 4) but excludes timing channels (Ex. 5) and empty observations (Ex. 6). Contrary to [26] there is no need for an artificial distinction between explicit (Ex. 1) and implicit flows (Ex. 2). In (3) and (4) both explicit and implicit flows are comprehended in exactly the same definition.

Notice that definition (4) of dependency is semantic-based and explicitly depends upon the program semantics. The notation $\mathsf{x} \overset{\ell}{\underset{\mathcal{S}^{+\infty}[\![\mathsf{P}]\!]}{\leadsto}} \mathsf{y}$ would be more precise.

Prefix versus maximal trace semantics based dependency The use of the prefix trace semantics $\mathcal{S}^{*}[\![\mathsf{P}]\!]$ is equivalent to that of the maximal trace semantics $\mathcal{S}^{+\infty}[\![\mathsf{P}]\!]$ in the definition (4) of dependency. This is formally stated by the following

Lemma 1 (Value dependency for finite prefix traces).

$$\mathsf{x} \leadsto^{\ell}_{\mathsf{P}} \mathsf{y} = \big(\mathcal{S}^{*}[\![\mathsf{P}]\!] \in \mathcal{D}^{\ell}\langle \mathsf{x}, \mathsf{y}\rangle\big).$$ □

Value dependency abstraction The value dependency abstraction of a semantic property $\mathcal{S} \in \wp(\wp(\mathbb{T}^{+} \times \mathbb{T}^{+\infty}))$ is

$$\alpha^{\mathrm{d}}(\mathcal{S})\ell \triangleq \{\langle \mathsf{x}, \mathsf{y}\rangle \mid \mathcal{S} \subseteq \mathcal{D}^{\ell}\langle \mathsf{x}, \mathsf{y}\rangle\} \tag{5}$$

Lemma 2. *There is a Galois connection $\langle \wp(\wp(\mathbb{T}^{+} \times \mathbb{T}^{+\infty})), \subseteq\rangle \underset{\alpha^{\mathrm{d}}}{\overset{\gamma^{\mathrm{d}}}{\leftrightarrows}} \langle \mathbb{P}^{\mathrm{d}}, \dot{\supseteq}\rangle$ where $\mathbb{P}^{\mathrm{d}} \triangleq \mathbb{L} \to \wp(\mathbb{V} \times \mathbb{V})$ is ordered pointwise and the concretization of a dependency property $\mathbf{D}$ is*

$$\gamma^{\mathrm{d}}(\mathbf{D}) \triangleq \bigcap_{\ell \in \mathbb{L}} \bigcap_{\langle \mathsf{x}, \mathsf{y}\rangle \in \mathbf{D}(\ell)} \mathcal{D}^{\ell}\langle \mathsf{x}, \mathsf{y}\rangle$$ □

The intuition is that the more semantics $\mathscr{S}$ have semantic property $\mathcal{S}$, the less dependencies can be found *i.e.* $\mathcal{S} \subseteq \mathcal{S}' \Rightarrow \alpha^{\mathrm{d}}(\mathcal{S})\ell \supseteq \alpha^{\mathrm{d}}(\mathcal{S}')\ell$. This is because the dependencies must exist for all semantics $\mathscr{S}$ having semantic property $\mathcal{S}$. Otherwise stated, the less dependencies you consider, the more semantics will exactly have these dependencies.

This is different from the observation than larger semantics have more dependencies $\mathscr{S} \subseteq \mathscr{S}' \Rightarrow \alpha^{\mathrm{d}}(\{\mathscr{S}\})\ell \subseteq \alpha^{\mathrm{d}}(\{\mathscr{S}'\})\ell$ since $\mathscr{S} \in \mathscr{D}\ell\langle \mathsf{x}, \mathsf{y}\rangle \Rightarrow \mathscr{S}' \in \mathscr{D}\ell\langle \mathsf{x}, \mathsf{y}\rangle$.

Value dependency semantics is an abstraction of the collecting trace semantics.

Corollary 1 (Value dependency for finite prefix traces).

$$\lambda\, \ell \bullet \{\langle \mathsf{x}, \mathsf{y}\rangle \mid \mathsf{x} \rightsquigarrow^{\ell}_{\mathrm{p}} \mathsf{y}\} = \alpha^{\mathrm{d}}(\{\mathscr{S}^{+\infty}[\![\mathsf{P}]\!]\}) = \alpha^{\mathrm{d}}(\{\mathscr{S}^{*}[\![\mathsf{P}]\!]\}) \qquad \square$$

Exact, definite, and potential value dependency semantics The exact value dependency semantics $\overline{\mathscr{S}}^{\mathrm{diff}}$ abstracts the maximal trace semantics, or equivalently, by Lem. 1, the prefix trace semantics by the dependency abstraction. By Rice theorem [55], $\{\mathscr{S}^{*}[\![\mathsf{S}]\!]\}$ is not computable so $\overline{\mathscr{S}}^{\mathrm{diff}}$ is not computable in this way. Therefore, static analysis must content itself with approximations (or unsoundness that we disapprove of). There are two possibilities. Definite value dependency is an under-approximation of value dependency (so $\varnothing$ is a correct under-approximation). Potential value dependency is an over-approximation of value dependency (so $\mathbb{V} \times \mathbb{V}$ is a correct over-approximation). Formally,

$$\begin{aligned}
\overline{\mathscr{S}}^{\mathrm{diff}}[\![\mathsf{S}]\!] &\triangleq \alpha^{\mathrm{d}}(\{\mathscr{S}^{+\infty}[\![\mathsf{S}]\!]\}) = \alpha^{\mathrm{d}}(\{\mathscr{S}^{*}[\![\mathsf{S}]\!]\}) && \text{exact dependency} \\
\overline{\mathscr{S}}^{\mathrm{diff}}_{\forall}[\![\mathsf{S}]\!] &\dot{\subseteq} \alpha^{\mathrm{d}}(\{\mathscr{S}^{+\infty}[\![\mathsf{S}]\!]\}) && \text{definite dependency} \\
\alpha^{\mathrm{d}}(\{\mathscr{S}^{+\infty}[\![\mathsf{S}]\!]\}) &\dot{\subseteq} \overline{\mathscr{S}}^{\mathrm{diff}}_{\exists}[\![\mathsf{S}]\!] && \text{potential dependency} \qquad (6)
\end{aligned}$$

We choose potential value dependency, which is an over-approximation of value dependency needed *e.g.* in compilation or security, looking for more dependencies than there are actually.

5 Calculational Design of the Structural Static Potential Value Dependency Analysis

Value dependency abstract domain An abstract property $\mathbf{D} \in \mathbb{P}^{\mathrm{d}}$ of the value dependency abstract domain $\mathbb{P}^{\mathrm{d}}$ tracks at each program point in $\ell \in \mathbb{L}$ the flows $\langle \mathsf{x}, \mathsf{y}\rangle \in \mathbf{D}\ell$ the initial value of x to the value of y at ℓ, that is $\mathsf{x} \rightsquigarrow^{\ell} \mathsf{y}$.

$$\mathbf{D} \in \mathbb{P}^{\mathrm{d}} \triangleq \mathbb{L} \to \wp(\mathbb{V} \times \mathbb{V}) \qquad (7)$$

$\langle \mathbb{P}^{\mathrm{d}}, \dot{\subseteq}, \dot{\bot}, \dot{\top}, \dot{\cap}, \dot{\cup}\rangle$ is a finite complete lattice partially ordered by pointwise subset inclusion $\dot{\subseteq}$. As in [26], values of variables are not taken into account in this abstraction. The Ex. 7 below shows that this introduces imprecision. This imprecision can be recovered by a reduced product [23,24] with a relational value analysis, which is an orthogonal problem.

Example 7 (structural compositionality). In the following statement, x and y at ℓ_1 depend on x at ℓ_0.

ℓ_0 y = x ;

ℓ_1 /* $x = x_0, y = y_0$ */

In the following statement, x and y at ℓ_2 depend on x at ℓ_1.

ℓ_1 y = y−x ;

ℓ_2 /* $x = x_0, y = y_0 - x_0$ */

In the sequential composition of the two statements

ℓ_0 y = x ; /* $x = x_0, y = y_0$ */

ℓ_1 y = y−x ; /* $x = x_0, y = x_0$ */

ℓ_2 /* $x = x_0, y = 0$ */

y at ℓ_2 depends on x at ℓ_1 which depends on x at ℓ_0 so, by composition, y at ℓ_2 depends on x at ℓ_0. However, y = 0 at ℓ_2 so y at ℓ_2 does not depend on x at ℓ_0.

For a more precise analysis, the reduced product of the dependency analysis and the linear equality analysis [40] will find $\exists x'_0, y'_0 \,.\, x'_0 = x_0 \wedge y'_0 = x_0 \wedge x = x'_0, y = y'_0 - x'_0$, that is, by projection, $x = x_0, y = 0$ so y at ℓ_2 does not depend on x at ℓ_0. □

Value dependency abstract semantics Whenever some term is not computable because it uses values, the calculational design of the potential value dependency semantics $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}]\!]$ will over-approximated it (as required by (6)). Therefore $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}]\!]$ is sound by construction. Besides the reduction with abstractions of values, this calculational design of $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}]\!]$ shows that it is possible to improve the precision of the analysis by taking the symbolic constancy of expressions into account.

Theorem 1. *For all program components* S*, the abstract value dependency semantics* $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}]\!]$ *defined by* (8) *to* (17) *is sound as specified by* (6) *of potential dependency.*

We obtain an abstract semantics operating by structural induction and computing fixpoints in a finite domain. It is therefore computable and directly yields an effective algorithm. We show the calculational design for the assignment, the other cases are similar.

- The **abstract potential dependency semantics at a statement** S **which is <u>not</u> an iteration**, variables have their initial value so only depend on themselves. (For loops (17) more dependencies may originate from the iterations.)

$$\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}]\!]\ \mathsf{at}[\![\mathsf{S}]\!] \triangleq \mathbb{1}_{\mathcal{V}} \tag{8}$$

where $\mathbb{1}_S \triangleq \{\langle \mathsf{x}, \mathsf{x}\rangle \mid \mathsf{x} \in S\}$ is the identity relation on set S and $\mathcal{V}$ is the set of program variables.

• The **abstract potential dependency semantics outside a statement**, there is no possible potential dependency since executions never reach that point.

$$\ell \notin \mathsf{labs}[\![\mathtt{S}]\!] \Rightarrow \overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell \triangleq \varnothing \tag{9}$$

• The **abstract potential dependency semantics after an assignment** $\mathtt{S} ::= \mathtt{x = A\ ;}$, the unmodified variables $\mathtt{y} \neq \mathtt{x}$ depend upon their initial value at $\mathsf{at}[\![\mathtt{S}]\!]$. The assigned-to variable $\mathtt{x}$ depends on $\overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!]$ defined as the variables on which the assigned expression $\mathtt{A}$ does depend.

$$\begin{array}{rcl}
\overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell & \triangleq & (\!|\, \ell = \mathsf{at}[\![\mathtt{S}]\!] \mathbin{?} \mathbb{1}_{\mathscr{V}} \\
& & \;\;|\!|\, \ell = \mathsf{aft}[\![\mathtt{S}]\!] \mathbin{?} \{\langle \mathtt{y}, \mathtt{x}\rangle \mid \mathtt{y} \in \overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!]\} \cup \{\langle \mathtt{y}, \mathtt{y}\rangle \mid \mathtt{y} \neq \mathtt{x}\} \\
& & \;\;\mathbin{:} \varnothing \,|\!) \\
\overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!] & \triangleq & \{\mathtt{y} \mid \exists \rho \in \mathbb{E}\mathscr{v} \,.\, \exists \nu \in \mathbb{V} \,.\, \mathscr{A}[\![\mathtt{A}]\!]\rho \neq \mathscr{A}[\![\mathtt{A}]\!]\rho[\mathtt{y} \leftarrow \nu]\} \subseteq \mathbb{vars}[\![\mathtt{A}]\!]
\end{array} \tag{10}$$

The functional dependency $\overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!]$ of expression $\mathtt{A}$ is traditionally over-approximated syntactically by the set of variables $\mathbb{vars}[\![\mathtt{A}]\!]$ of this expression $\mathtt{A}$ [68,26]. This is very coarse since *e.g.* if $\mathtt{A}$ is constant (such as $\mathtt{y = x - x\ ;}$), $\overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!]$ is empty. For a trivial improvement, we can define

$$\overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{1}]\!] \triangleq \varnothing \qquad \overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{x}]\!] \triangleq \{\mathtt{x}\} \qquad \overline{\mathscr{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}_1 - \mathtt{A}_2]\!] \triangleq \{\mathtt{y} \in \mathbb{vars}[\![\mathtt{A}_1]\!] \cup \mathbb{vars}[\![\mathtt{A}_2]\!] \mid \mathtt{A}_1 \neq \mathtt{A}_2\}.$$

The analysis looks quite imprecise. Further precision can be obtained by a reduced product with a value analysis, as examplified in Ex. 7 and later discussed in Section **6**.

The interest of the proof of (10) is to show that the value dependency algorithm follows by calculus from the trace semantics of [20, Section **2**] and the abstraction (5). By varying the semantics this can be applied to other languages. By varying the abstraction, one can consider the different variants of dependency. In another context of safety analysis, such proofs have been shown to be machine checkable [39] and hopefully, in the future, automatisable.

Proof (of (10)). The cases $\ell = \mathsf{at}[\![\mathtt{S}]\!]$ was handled in (8) and $\ell \notin \mathsf{labs}[\![\mathtt{S}]\!]$ in (9). It remains the case $\ell = \mathsf{aft}[\![\mathtt{S}]\!]$.

$$\begin{array}{ll}
& \alpha^{\mathsf{d}}(\{\mathscr{S}^{+\infty}[\![\mathtt{S}]\!]\})\,\mathsf{aft}[\![\mathtt{S}]\!] \\
= & \alpha^{\mathsf{d}}(\{\mathscr{S}^{*}[\![\mathtt{S}]\!]\})\,\mathsf{aft}[\![\mathtt{S}]\!] \qquad \text{(Lem. 1)} \\
= & \{\langle \mathtt{x}', \mathtt{y}\rangle \mid \mathscr{S}^{*}[\![\mathtt{S}]\!] \in \mathcal{D}(\mathsf{aft}[\![\mathtt{S}]\!])\langle \mathtt{x}', \mathtt{y}\rangle\} \qquad \text{(def. (5) of } \alpha^{\mathsf{d}} \text{ and def. } \subseteq) \\
= & \{\langle \mathtt{x}', \mathtt{y}\rangle \mid \exists \langle \pi_0, \pi_1\rangle, \langle \pi'_0, \pi'_1\rangle \in \mathscr{S}^{*}[\![\mathtt{S}]\!] \,.\, \forall \mathtt{z} \in \mathscr{V} \setminus \{\mathtt{x}'\} \,.\, \varrho(\pi_0)\mathtt{z} = \varrho(\pi'_0)\mathtt{z} \wedge \\
& \mathsf{diff}(\mathsf{seqval}[\![\mathtt{y}]\!](\mathsf{aft}[\![\mathtt{S}]\!])(\pi_0, \pi_1), \mathsf{seqval}[\![\mathtt{y}]\!](\mathsf{aft}[\![\mathtt{S}]\!])(\pi'_0, \pi'_1))\} \\
& \qquad \text{(def. } \in \text{ and (3) of } \mathcal{D}^{\ell}\langle \mathtt{x}', \mathtt{y}\rangle) \\
= & \{\langle \mathtt{x}', \mathtt{y}\rangle \mid \exists \langle \pi_0, \pi_1\rangle, \langle \pi'_0, \pi'_1\rangle \in \{\langle \pi\mathsf{at}[\![\mathtt{S}]\!], \mathsf{at}[\![\mathtt{S}]\!] \xrightarrow{\mathtt{x} = \mathscr{A}[\![\mathtt{A}]\!]\varrho(\pi\mathsf{at}[\![\mathtt{S}]\!])} \mathsf{aft}[\![\mathtt{S}]\!]\rangle \mid \pi\mathsf{at}[\![\mathtt{S}]\!] \in \mathbb{T}^{+}\} \,.\, \forall \mathtt{z} \in \mathscr{V} \setminus \{\mathtt{x}'\} \,.\, \varrho(\pi_0)\mathtt{z} = \varrho(\pi'_0)\mathtt{z} \wedge \\
& \mathsf{diff}(\mathsf{seqval}[\![\mathtt{y}]\!](\mathsf{aft}[\![\mathtt{S}]\!])(\pi_0, \pi_1), \mathsf{seqval}[\![\mathtt{y}]\!](\mathsf{aft}[\![\mathtt{S}]\!])(\pi'_0, \pi'_1))\}
\end{array}$$

$\wr$def. ([20].3) of the assignment prefix finite trace semantics$\int$

$$= \{\langle \mathsf{x}', \ \mathsf{y}\rangle \mid \exists \langle \pi_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle, \langle \pi'_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle \ . \ \forall \mathsf{z} \in \mathbb{V} \setminus \{\mathsf{x}'\} \ . \ \varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z} = \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z} \wedge \mathsf{diff}(\mathsf{seqval}[\![\mathsf{y}]\!](\mathsf{aft}[\![\mathsf{S}]\!])(\pi_0 \mathsf{at}[\![\mathsf{S}]\!], \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]), \mathsf{seqval}[\![\mathsf{y}]\!](\mathsf{aft}[\![\mathsf{S}]\!])(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!], \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]))\} \quad \wr\text{def. } \in\int$$

$$= \{\langle \mathsf{x}', \ \mathsf{y}\rangle \mid \exists \langle \pi_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle, \langle \pi'_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle \ . \ (\forall \mathsf{z} \in \mathbb{V} \setminus \{\mathsf{x}'\} \ . \ \varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z} = \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z}) \wedge \mathsf{diff}(\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!])\mathsf{y}, \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!])\mathsf{y})\}$$

$\wr$def. (1) of the future $\mathsf{seqval}[\![\mathsf{y}]\!]\int$

$$= \{\langle \mathsf{x}', \ \mathsf{y}\rangle \mid \exists \langle \pi_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle, \langle \pi'_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle \ . \ (\forall \mathsf{z} \in \mathbb{V} \setminus \{\mathsf{x}'\} \ . \ \varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z} = \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z}) \wedge ((\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{y} \neq \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{y}) \vee (\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{y} = \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{y} \wedge \varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!])\mathsf{y} \neq \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!])\mathsf{y})\}$$

$\wr(2)$ so that $\mathsf{diff}(a \cdot b,\ c \cdot d)$ if and only if (1) $a \neq c$ or (2) $a = c \wedge b \neq d$.$\int$

$$= \{\langle \mathsf{x}', \ \mathsf{y}\rangle \mid \exists \langle \pi_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle, \langle \pi'_0 \mathsf{at}[\![\mathsf{S}]\!], \ \mathsf{at}[\![\mathsf{S}]\!] \xrightarrow{\mathsf{x}=\mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])} \mathsf{aft}[\![\mathsf{S}]\!]\rangle \ . \ (\forall \mathsf{z} \in \mathbb{V} \setminus \{\mathsf{x}'\} \ . \ \varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z} = \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z}) \wedge ((\mathsf{y} = \mathsf{x}') \vee (\mathsf{y} = \mathsf{x} \wedge \mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!]) \neq \mathscr{A}[\![\mathsf{A}]\!]\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])))\} \quad \wr\text{def. (2) of } \varrho\int$$

$$\subseteq \{\langle \mathsf{x}', \ \mathsf{y}\rangle \mid ((\mathsf{y} = \mathsf{x}') \vee (\mathsf{y} = \mathsf{x} \wedge \exists \rho, \nu \ . \ \mathscr{A}[\![\mathsf{A}]\!]\rho \neq \mathscr{A}[\![\mathsf{A}]\!]\rho[\mathsf{x}' \leftarrow \nu]))\} \qquad (11)$$

$\wr$letting $\rho = \varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])$ and $\nu = \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])(\mathsf{x}')$ so that $\forall \mathsf{z} \in \mathbb{V} \setminus \{\mathsf{x}'\}$. $\varrho(\pi_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z} = \varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!])\mathsf{z}$ implies that $\varrho(\pi'_0 \mathsf{at}[\![\mathsf{S}]\!]) = \rho[\mathsf{x}' \leftarrow \nu]$.$\int$

$$= \{\langle \mathsf{x}', \ \mathsf{x}'\rangle \mid \mathsf{x}' \neq \mathsf{x}\} \cup \{\langle \mathsf{x}', \ \mathsf{x}\rangle \mid \exists \rho, \nu \ . \ \mathscr{A}[\![\mathsf{A}]\!]\rho \neq \mathscr{A}[\![\mathsf{A}]\!]\rho[\mathsf{x}' \leftarrow \nu]\} \quad \wr\text{case analysis}\int$$

$$= \{\langle \mathsf{x}', \ \mathsf{x}'\rangle \mid \mathsf{x}' \neq \mathsf{x}\} \cup \{\langle \mathsf{x}', \ \mathsf{x}\rangle \mid \mathsf{x}' \in \overline{\mathscr{S}}^{\,\mathsf{diff}}_{\exists}[\![\mathsf{A}]\!]\}$$

$\wr$by defining the functional dependency of an expression A as $\overline{\mathscr{S}}^{\,\mathsf{diff}}_{\exists}[\![\mathsf{A}]\!] \triangleq \{\mathsf{x}' \mid \exists \rho, \nu \ . \ \mathscr{A}[\![\mathsf{A}]\!]\rho \neq \mathscr{A}[\![\mathsf{A}]\!]\rho[\mathsf{x}' \leftarrow \nu]\}$ in (10)$\int$ □

Equality holds in (11) if every environment is computable *i.e.* $\forall \rho \in \mathbb{E}\mathsf{v} \ . \ \exists \pi \in \mathbb{T}^+ \ . \ \varrho(\pi) = \rho$. Then imprecision comes only from $\overline{\mathscr{S}}^{\,\mathsf{diff}}_{\exists}[\![\mathsf{A}]\!]$ *i.e.* when it is impossible to evaluate A in all possible environments. Notice that a reduced product with a reachability analysis providing an invariant $\mathsf{at}[\![\mathsf{S}]\!]$ will limit the possible values of the environments ρ and $\rho[\mathsf{x}' \leftarrow \nu]$ in $\overline{\mathscr{S}}^{\,\mathsf{diff}}_{\exists}[\![\mathsf{A}]\!]$ and therefore make the dependency analysis more precise.

- The **abstract potential value dependency semantics of a conditional statement** $\mathtt{S}$ ::= $\mathtt{if\ (B)\ S}_t$ is specified in (12) below. It was discovered by calculational design. (The left restriction $r \rceil S$ of a relation $r \in \wp(S_1 \times S_2)$ to a set S is $\{\langle x,\ y\rangle \in r \mid x \in S\}$.)

$$\begin{aligned}
\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell \triangleq \big(\!\big[\, & \ell = \mathsf{at}[\![\mathtt{S}]\!] \;?\; \mathbb{1}_{\mathcal{V}} && \text{(a)} \quad (12)\\
\big[\!\big[\, & \ell \in \mathsf{in}[\![\mathtt{S}_t]\!] \;?\; \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}_t]\!]\,\ell \,\big]\!\big] \,\mathsf{nondet}(\mathtt{B},\mathtt{B}) && \text{(b)}\\
\big[\!\big[\, & \ell = \mathsf{aft}[\![\mathtt{S}]\!] \;?\; \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}_t]\!]\,\mathsf{aft}[\![\mathtt{S}_t]\!] \,\big]\!\big] \,\mathsf{nondet}(\mathtt{B},\mathtt{B}) && \text{(c.1)}\\
& \qquad \cup\, \mathbb{1}_{\mathcal{V}} \,\big]\!\big] \,\mathsf{nondet}(\neg\mathtt{B},\neg\mathtt{B}) && \text{(c.2)}\\
& \qquad\quad \cup\, \mathsf{nondet}(\neg\mathtt{B},\neg\mathtt{B}) \times \mathsf{mod}[\![\mathtt{S}_t]\!] && \text{(c.3)}\\
& \;\wr\, \varnothing \,\big)\!\big] && \text{(d)}
\end{aligned}$$

On entry (12.a), which is an instance of (8), variables in $\mathcal{V}$ only depend upon themselves as specified by the identity relation $\mathbb{1}_{\mathcal{V}}$.

The reasoning in (12.b) is that if a variable $\mathtt{y}$ depends at ℓ on the initial value of a variable $\mathtt{x}$ at $\mathsf{at}[\![\mathtt{S}_t]\!]$, it depends in the same way on that initial value of the variable $\mathtt{x}$ at $\mathsf{at}[\![\mathtt{S}]\!]$ since the test $\mathtt{B}$ has no side effect. However, (12.b) also takes into account that if $\mathtt{S}_t$ can only be reached for a unique value of the variable $\mathtt{x}$ and the branch is not taken for all other values of $\mathtt{x}$ then the variable $\mathtt{y}$ does not depend on $\mathtt{x}$ in $\mathtt{S}_t$ since empty observations are disallowed by the abstraction (5) using the definition (2) of diff.

[Non-]determinacy $\mathsf{det}(\mathtt{B}_1,\mathtt{B}_2)$ [$\mathsf{nondet}(\mathtt{B}_1,\mathtt{B}_2)$] is defined *s.t.*

$$\begin{aligned}
&\mathsf{det}(\mathtt{B}_1,\mathtt{B}_2) \subseteq \{\mathtt{x} \mid \forall \rho,\rho' \,.\, (\boldsymbol{\mathcal{B}}[\![\mathtt{B}_1]\!]\rho \wedge \boldsymbol{\mathcal{B}}[\![\mathtt{B}_2]\!]\rho') \Rightarrow (\rho(\mathtt{x}) = \rho'(\mathtt{x}))\} && (13)\\
&\mathsf{nondet}(\mathtt{B}_1,\mathtt{B}_2) \supseteq \mathcal{V} \setminus \mathsf{det}(\mathtt{B}_1,\mathtt{B}_2)
\end{aligned}$$

So if $\mathtt{x} \in \mathsf{det}(\mathtt{B}_1,\mathtt{B}_2)$ in (13) then $\mathtt{B}_1$ and $\mathtt{B}_2$ can both be true for at most one value of $\mathtt{x}$ (*e.g.* $\mathsf{det}(\mathtt{x==1},\mathtt{x==1}) = \{\mathtt{x}\}$ and $\mathsf{det}(\mathtt{x==1},\mathtt{x!=1}) = \varnothing$). It is under-approximated by $\varnothing$. Its complement $\mathsf{nondet}(\mathtt{B}_1,\mathtt{B}_2)$ in (13) is the set of variables for which $\mathtt{B}_1$ and $\mathtt{B}_2$ may both be true for different values of variable $\mathtt{x}$. It is over-approximated by all variables $\mathcal{V}$. A better solution is to use a reduced product with a value or symbolic constant propagation analysis as in Section **6**.

If $\mathtt{x} \notin \mathsf{nondet}(\mathtt{B},\mathtt{B})$ in (12.b) then $\mathtt{x} \in \mathsf{det}(\mathtt{B},\mathtt{B})$ so the value of $\mathtt{x}$ is constant in $\mathtt{S}_t$ so no variable $\mathtt{y}$ in $\mathtt{S}_t$ can depend on $\mathtt{x}$. For example dependency at ℓ in `if (x == 1) { y = x ;` ℓ `}` is the same as `x = 1 ;y = x ;` ℓ, which is the same as `y = 1 ;` ℓ so $\mathtt{y}$ does not depends on $\mathtt{x}$ at ℓ.

(12.c) determines dependencies after $\mathtt{S}$ so compare two possible executions of that statement. In case (12.c.1) both executions go through the true branch. In case (12.c.2) both executions go through the false branch, while in case (12.c.3) the executions take different branches.

In case (12.c.1) when the test is true $\mathtt{tt}$ for both executions, the executions of the true branch $\mathtt{S}_t$ terminate and control after $\mathtt{S}_t$ reaches the program point after $\mathtt{S}$ (recall that $\mathsf{aft}[\![\mathtt{S}_t]\!] = \mathsf{aft}[\![\mathtt{S}]\!]$). The dependencies after $\mathtt{S}_t$ propagate after $\mathtt{S}$ but only in case of non-determinism, *e.g.* for variables that are not constant.

The second case in (12.c.2) is for those executions for which the test $\mathtt{B}$ is false $\mathtt{ff}$. Variables depend on themselves $\mathsf{at}[\![\mathtt{S}]\!]$ and control moves to $\mathsf{aft}[\![\mathtt{S}]\!]$ so that dependencies are the same there, but only for variables that can reach $\mathsf{aft}[\![\mathtt{S}]\!]$ with different values on different executions as indicated by the restriction to $\mathsf{nondet}(\neg\mathtt{B},\neg\mathtt{B})$.

The third case in (12.c.3) is for pairs of executions, one through the true branch and the other through the false branch. In that case $\mathtt{y}$ depends on $\mathtt{x}$ only if $\mathtt{x}$ does not force execution to always take the same branch, meaning that $\mathtt{x} \in \mathsf{nondet}(\neg\mathtt{B}, \neg\mathtt{B})$. If $\mathtt{y}$ is not modified by the execution through $\mathtt{S}_t$ then its value after $\mathtt{S}$ is always the same as its value $\mathsf{at}[\![\mathtt{S}]\!]$ (since $\mathtt{y}$ is not modified on the false branch either). In that case changing $\mathtt{y}$ $\mathsf{at}[\![\mathtt{S}]\!]$ would not change $\mathtt{y}$ after $\mathtt{S}$ so that, in that situation, $\mathtt{y}$ does not depend on $\mathtt{x}$. Therefore (12.c.3) requires that $\mathtt{y} \in \mathsf{mod}[\![\mathtt{S}_t]\!]$.

The variables $\mathsf{mod}[\![\mathtt{S}]\!]$ modified by a statement $\mathtt{S}$ are

$$\mathsf{mod}[\![\mathtt{S}]\!] \supseteq \{\mathtt{x} \mid \exists \pi_0, \pi_1 \,.\, \langle \pi_0\mathsf{at}[\![\mathtt{S}]\!], \mathsf{at}[\![\mathtt{S}]\!]\pi_1\mathsf{aft}[\![\mathtt{S}]\!]\rangle \in \mathcal{S}^{+\infty}[\![\mathtt{S}]\!] \\ \wedge \varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!]\pi_1\mathsf{aft}[\![\mathtt{S}]\!])\mathtt{x} \neq \varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!])\mathtt{x}\} \quad (14)$$

In the style of [26], a purely syntactic and very rough over-approximation would be

$$\mathsf{mod}[\![\mathtt{x = E\ ;}]\!] \triangleq \{\mathtt{x}\} \qquad \mathsf{mod}[\![\mathtt{if\ (B)}\ \mathtt{S}_t\ \mathtt{else}\ \mathtt{S}_f]\!] \triangleq \mathsf{mod}[\![\mathtt{S}_t]\!] \cup \mathsf{mod}[\![\mathtt{S}_f]\!] \quad (15)$$
$$\mathsf{mod}[\![\mathtt{\{\ Sl\ \}}]\!] \triangleq \mathsf{mod}[\![\mathtt{Sl}]\!] \qquad \mathsf{mod}[\![\mathtt{while\ (B)\ S}]\!] \triangleq \mathsf{mod}[\![\mathtt{if\ (B)\ S}]\!] \triangleq \mathsf{mod}[\![\mathtt{S}]\!]$$
$$\mathsf{mod}[\![\mathtt{Sl\ S}]\!] \triangleq \mathsf{mod}[\![\mathtt{Sl}]\!] \cup \mathsf{mod}[\![\mathtt{S}]\!] \qquad \mathsf{mod}[\![\mathtt{;}]\!] \triangleq \mathsf{mod}[\![\,\epsilon\,]\!] \triangleq \mathsf{mod}[\![\mathtt{break\ ;}]\!] \triangleq \varnothing$$

Again Section **6** applies. A reduced product with a reachability analysis would be more precise *e.g.* because the variable is constant on exit of $\mathtt{S}$ or a relational analysis such that linear equalities [40] shows that it is equal to its initial value.

Finally in case (12.d) the program point ℓ is not reachable in $\mathtt{S}$ so, as stated in (9) there is not dependency at ℓ originating from $\mathtt{S}$.

Example 8. Consider $\mathtt{S} ::= \ell\ \mathtt{L = H\ ;}\ell'$. We have $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell = \{\langle \mathtt{x}, \mathtt{x}\rangle \mid \mathtt{x} \in \mathbb{V}\}$ and $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell' = \{\langle \mathtt{H}, \mathtt{L}\rangle\} \cup \{\langle \mathtt{x}, \mathtt{x}\rangle \mid \mathtt{x} \in \mathbb{V} \setminus \{\mathtt{L}\}\}$.

We have $\mathsf{nondet}(\mathtt{H}, \mathtt{H}) = \mathsf{nondet}(\neg\mathtt{H}, \neg\mathtt{H}) = \{\mathtt{L}\}$ so that for the statement $\mathtt{S}' ::= \mathtt{\{\ if}\ \ell_1\ \mathtt{(H)}\ \ell_2\ \mathtt{L = H\ ;}\ell_3\ \mathtt{else}\ \ell_4\ \mathtt{L = H\ ;}\ell_5\ \mathtt{\}}\ell_6$, we have

$$\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}']\!]\,\ell_1 = \{\langle \mathtt{x}, \mathtt{x}\rangle \mid \mathtt{x} \in \mathbb{V}\}$$
$$\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}']\!]\,\ell_2 = \overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}']\!]\,\ell_4 = \{\langle \mathtt{x}, \mathtt{x}\rangle \mid \mathtt{x} \in \mathbb{V} \setminus \{\mathtt{H}\}\}$$
$$\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}']\!]\,\ell_3 = \overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}']\!]\,\ell_5 = \{\langle \mathtt{x}, \mathtt{x}\rangle \mid \mathtt{x} \in \mathbb{V} \setminus \{\mathtt{L}\}\}$$
$$\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}']\!]\,\ell_6 = \{\langle \mathtt{H}, \mathtt{L}\rangle\} \cup \{\langle \mathtt{x}, \mathtt{x}\rangle \mid \mathtt{x} \in \mathbb{V} \setminus \{\mathtt{L}\}\}$$

This is different and more precise than *e.g.* [5,26] since $\mathtt{L}$ does not depend on $\mathtt{H}$ at ℓ_3 and ℓ_5 since, by def. nondet, $\mathtt{L}$ is constant at ℓ_3 and ℓ_5. So this is equivalent to $\ell_2\ \mathtt{L = true\ ;}\ell_3$ and $\ell_4\ \mathtt{L = false\ ;}\ell_5$ which obviously would create no dependency. In contrast [5,26] maintain an imprecise control dependence context, denoting (a superset of) the variables that at least one test surrounding $\mathtt{S}$ depends on. Section **6** provides other examples of increased precision when taking values of variables into account. □

r • The **abstract potential dependency semantics of a statement list** $\mathtt{Sl} ::= \mathtt{Sl}'\ \mathtt{S}$ is

$$\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{Sl}]\!]\,\ell \triangleq \big(\!\big[\, \ell \in \mathsf{labs}[\![\mathtt{Sl}']\!] \;?\; \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{Sl}']\!]\,\ell \tag{16.a}$$

$$\big[\!\big[\, \ell \in \mathsf{labs}[\![\mathtt{S}]\!] \setminus \{\mathsf{at}[\![\mathtt{S}]\!]\} \;?\; \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{Sl}']\!]\,\mathsf{at}[\![\mathtt{S}]\!] \fatsemi \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell \tag{16.b}$$

$$\fatsemi\, \varnothing \,\big]\!\big)$$

where the composition $\fatsemi$ of relations is $r_1 \fatsemi r_2 \triangleq \{\langle x,\, y\rangle \mid \exists z \,.\, \langle x,\, z\rangle \in r_1 \wedge \langle z,\, y\rangle \in r_2\}$.

The first case (16.a) looks for dependencies at a program point ℓ inside $\mathtt{Sl}'$ so, by structural induction, this is $\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{Sl}']\!]\,\ell$.

The second case (16.b) looks for dependencies at a program point ℓ inside $\mathtt{S}$. We exclude the case $\ell = \mathsf{at}[\![\mathtt{S}]\!]$ since $\mathsf{at}[\![\mathtt{S}]\!] = \mathsf{aft}[\![\mathtt{Sl}']\!] \in \mathsf{labs}[\![\mathtt{Sl}']\!]$ so this case has already been handled in the previous case (16.a).

Otherwise in (16.b), a variable $\mathtt{y}$ at ℓ in $\mathtt{S}$ depends on the initial value of a variable $\mathtt{x}$ on entry $\mathsf{at}[\![\mathtt{Sl} ::= \mathtt{Sl}'\ \mathtt{S}]\!] = \mathsf{at}[\![\mathtt{Sl}']\!] \in \mathsf{labs}[\![\mathtt{Sl}']\!]$ if and only if $\mathtt{y}$ at ℓ in $\mathtt{S}$ depends on the initial value of some variable $\mathtt{z}$ on entry $\mathsf{at}[\![\mathtt{S}]\!] = \mathsf{aft}[\![\mathtt{Sl}']\!]$ of statement $\mathtt{S}$ and the value of $\mathtt{z}$ at that point depends on the initial value of a variable $\mathtt{x}$ on entry $\mathsf{at}[\![\mathtt{Sl}]\!]$ of the statement list $\mathtt{Sl}$. So there exists $\mathtt{z}$ such that $\langle \mathtt{y},\, \mathtt{z}\rangle \in \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{Sl}']\!]\,\mathsf{aft}[\![\mathtt{Sl}']\!] = \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{Sl}']\!]\,\mathsf{at}[\![\mathtt{S}]\!]$ and $\langle \mathtt{z},\, \mathtt{y}\rangle \in \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell$, meaning that $\langle \mathtt{x},\, \mathtt{y}\rangle \in \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{Sl}']\!]\,\mathsf{at}[\![\mathtt{S}]\!] \fatsemi \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell$ is in their composition. As shown by Ex. 7, the precision can be improved by a reduced product with a value analysis.

- The **abstract potential dependency semantics of an iteration statement** $\mathtt{S} ::= \mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b$ is the following

$$\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!]\,\ell' = (\mathsf{lfp}^{\dot{\subseteq}}\, \boldsymbol{\mathcal{F}}^{\mathsf{diff}}_{\exists}[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!])\,\ell' \tag{17}$$

$$\boldsymbol{\mathcal{F}}^{\mathsf{diff}}_{\exists}[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!]\,X\,\ell' =$$

$$\big(\!\big[\, \ell' = \ell \;?\; \mathbb{1}_{\mathbb{V}} \cup \big(X(\ell) \fatsemi (\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}_b]\!]\,\ell \rceil\, \mathsf{nondet}(\mathtt{B},\mathtt{B}))\big) \tag{a}$$

$$\big[\!\big[\, \ell' \in \mathsf{in}[\![\mathtt{S}_b]\!] \;?\; X(\ell) \fatsemi \big(\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}_b]\!]\,\ell' \rceil\, \mathsf{nondet}(\mathtt{B},\mathtt{B})\big) \tag{b}$$

$$\big[\!\big[\, \ell' = \mathsf{aft}[\![\mathtt{S}]\!] \;?\; X(\ell) \cup \big(X(\ell) \fatsemi (\mathbb{V} \times \mathsf{mod}[\![\mathtt{S}_b]\!])\big) \cup \tag{c}$$

$$X(\ell) \fatsemi \Big(\Big(\bigcup\nolimits_{\ell'' \in \mathsf{brks\text{-}of}[\![\mathtt{S}_b]\!]} \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}_b]\!]\,\ell''\Big) \rceil\, \mathsf{nondet}(\mathtt{B},\mathtt{B})\Big)$$

$$\fatsemi\, \varnothing \,\big]\!\big) \tag{d}$$

Since $\boldsymbol{\mathcal{F}}^{\mathsf{diff}}_{\exists}[\![\mathtt{S}]\!] \in \mathbb{P}^{\mathsf{d}} \to \mathbb{P}^{\mathsf{d}}$ is $\dot{\subseteq}$-monotone and the abstract domain $\langle \mathbb{P}^{\mathsf{d}},\, \dot{\subseteq},\, \lambda\,\ell \bullet \varnothing,\, \lambda\,\ell \bullet \mathbb{V} \times \mathbb{V},\, \dot{\cup},\, \dot{\cap}\rangle$ in (7) is a complete lattice, the least fixpoint $\mathsf{lfp}^{\dot{\subseteq}}\, \boldsymbol{\mathcal{F}}^{*}[\![\mathtt{S}]\!]$ of $\boldsymbol{\mathcal{F}}^{*}[\![\mathtt{S}]\!]$ exists by Tarski's fixpoint theorem [62]. Moreover, since $\mathbb{P}^{\mathsf{d}}$ is finite (at least when considering only the program labels $\mathbb{L}$ and variables $\mathbb{V}$ occurring in a program), the abstract properties of $\mathbb{P}^{\mathsf{d}}$ have a finite computer memory representation and the limit of iterates [22] can be computed in finitely many iterations, which yields an effective static analysis algorithm.

$\mathsf{lfp}^{\dot{\subseteq}}\, \boldsymbol{\mathcal{F}}^{\mathsf{diff}}_{\exists}[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!]$ is the least solution to the system of equations

$$\begin{cases} X(\ell') = \boldsymbol{\mathcal{F}}^{\mathsf{diff}}_{\exists}[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!]\,X\,\ell' \\ \ell' \in \mathbb{L} \end{cases}$$

which can be understood as follows.

(a) On loop entry the variables depend on themselves. After several iterations the dependency is the composition of the dependencies of the previous iterations $X(\ell)$ composed with those $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}_b]\!]\,\ell$ created by one more iteration. The composition $\fatsemi$ is that used for a list of statements in (16). The restriction $\rceil\mathsf{nondet}(\mathtt{B},\mathtt{B})$ eliminates dependencies on variables with a single possible value on loop [re-]entry;

(b) The initial value of a variable $\mathtt{x}$ on loop entry flows to a variable $\mathtt{z}$ at $\ell' \in \mathsf{in}[\![\mathsf{S}_b]\!]$ in the loop body if and only if the initial value of a variable $\mathtt{x}$ on loop entry flows to some variable $\mathtt{y}$ at loop entry ℓ after 0 or more iterations (so $\langle \mathtt{x}, \mathtt{y}\rangle \in X(\ell)$) and the value of variable $\mathtt{y}$ at the loop body flows to $\mathtt{z}$ at ℓ' (so $\langle \mathtt{y}, \mathtt{z}\rangle \in \overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}_b]\!]\,\ell'$). Moreover the restriction $\mathsf{nondet}(\mathtt{B},\mathtt{B})$ eliminates the case when $\mathtt{x}$ is constant after the loop test $\mathtt{B}$. (The case when $\mathtt{y}$ is constant after the loop test $\mathtt{B}$ has been recursively eliminated by $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}_b]\!]\,\ell'$));

(c) This case determines the dependencies on loop exit $\ell' = \mathsf{aft}[\![\mathsf{S}]\!]$, not knowing the values of variables, so the number of iterations and how the loop is exited is unknown. Therefore all cases must be considered.
- The term $X(\ell)$ (where $\ell \triangleq \mathsf{at}[\![\mathsf{S}]\!] = \mathsf{aft}[\![\mathsf{S}_b]\!]$) corresponds to the case when the loop is entered and iterated 0 or more times before exiting so either the loop is never entered so each variable depends on itself by (17.a) or the dependencies on exit of the loop are those after the last iteration of the body;
- The term $\left(X(\ell) \fatsemi (\mathcal{V} \times \mathsf{mod}[\![\mathsf{S}_b]\!])\right)$ covers dependencies originating from two executions decided by the initial values of a variable $\mathtt{x}$ such that in one case the loop is entered and exited and for another value it is immediately exited. The variables $\mathtt{y}$ modified in the loop body depend on $\mathtt{x}$, as was the case in (12.c.3) for the conditional;
- The term $\bigcup_{\ell'' \in \mathsf{brks\text{-}of}[\![\mathsf{S}]\!]} (X(\ell) \fatsemi \overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathsf{S}_b]\!]\,\ell'')$ propagates the dependencies at the `break ;` statements within the loop body to the break point $\mathsf{aft}[\![\mathsf{S}]\!]$ after the loop;
- The term (17.c) can be refined to take the test determinism into account more precisely, by eliminating those cases for which it is sure that no two distinct executions can be found in the definition of dependency;

(d) The iteration statement `while` $^{\ell}$ `(B)` S_b introduces no dependency outside its reachable points.

• The remaining cases of the conditional with alternative, empty statement list, skip, break, and compound statements are similar.

6 Reduced product with a relational value analysis

$\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!]$ in (10) handles the case $\overline{\mathcal{S}}^{\mathsf{diff}}_{\exists}[\![\mathtt{x} - \mathtt{x}]\!] = \varnothing$ while $\mathsf{vars}[\![\mathtt{x} - \mathtt{x}]\!] = \{\mathtt{x}\}$. As shown in Ex. 7, even more precision can be achieved by considering reachable environments only. The abstraction $\alpha_r(\mathcal{S}[\![\mathsf{S}]\!])\ell$ of the trace semantics $\mathcal{S}[\![\mathsf{S}]\!]$ of a program component S by the classical relation abstraction

$$\alpha_r(\mathcal{S}) \triangleq \lambda\, \ell \in \mathbb{L} \bullet \{\langle \varrho(\pi_0),\, \varrho(\pi_0 \frown \pi_1\ell)\rangle \mid \langle \pi_0,\, \pi_1\ell\rangle \in \mathcal{S}\}$$

provides a relation between the initial value of variables and their value at a program point ℓ of $\mathtt{S}$ (the relation is empty if ℓ is not in $\mathtt{S}$).

Then the dependency analysis can be refined using this relational value information.

- For the assignment (10), the imprecision is only due to the term $\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!]$ because it is impossible to evaluate the arithmetic expression $\mathtt{A}$ in all reachable environments on entry of the assignment (see step (11) of the calculation design). However, a relational value static analysis can provide relevant information.
  For example, using a constant propagation either cartesian [41,67] or relational in [40,42,52], or a zone/octagon analysis [50], $\mathtt{y} \in \overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}_1 - \mathtt{A}_2]\!]$ only if this analysis cannot prove that $\mathtt{A}_1 - \mathtt{A}_2$ is constant.
- For the conditional (12) and the iteration (17), the relational value static analysis can provide relevant information on non-determinacy nondet in (13).
- For sequential composition, conditionals, and iteration, it is also possible to refine the calculational design to improve compositionality. For example, in $\mathtt{if\ (H)\ L=X;\ else\ L=X;}$, the above relational value analyzes yield $\mathtt{L=X}_0$ on exit of the conditional so $\mathtt{L}$ does not depend on $\mathtt{H}$.

This is better implemented by a reduced product [23,69,19] and side conditions in the dependency analysis (such as $\overline{\boldsymbol{\mathcal{S}}}^{\mathsf{diff}}_{\exists}[\![\mathtt{A}]\!]$ and nondet) refining dependencies using relational value information provided by the other domains in the product. This separation of concerns greatly simplifies the design of the analysis [25].

7 Examples of derived dependency semantics and analyzes

Independence Definite independence is the complement of potential dependency. [53,4,5] introduced a Hoare-like logic to statically check independences. It also takes nontermination into account so relies on a different definition of $\neg\mathcal{D}^{\ell}\langle \mathtt{x}, \mathtt{y}\rangle$. It is recognized that definite independence is an abstract interpretation but this is not used to design the logic which remains empirical.

Abstract non-interference/dependency The abstraction $\alpha^{\mathrm{d}}(\mathcal{S})$ of the semantic property $\mathcal{S}$ in (5) is meaningful for any semantic property $\mathcal{S}$, including abstract ones, as considered in abstract non-interference/dependency [29]. Given a structural semantic definition of this abstract property, the principle of design by calculational design of the abstract dependency remains the same.

Forward and backward dependency Dependency information is useful for program slicing [68]. The semantics ([20].3)—([20].9) considered in Section **2** is forward, defining the continuation in a program component of an initialization

computation ending on entry of that program component. This forward dependency is adequate for forward slicing [5]. The dependency abstraction may be applied to a backward semantics defining the reachability in a program component of an finalization computation starting at that end of a program component or on a break. This backward dependency would certainly be more useful as a basis for slicing [68], or abstract slicing [37,57,49].

Dye instrumented semantics By analogy with dye-tracer tests in hydrology to determine the possible origins of spring discharges or resurgences by water source coloring and flow tracing [43], it has been suggested to decorate the initial values of variables with labels such as color annotations and to track their diffusion and mixtures to determine dependencies [17]. This postulated definition of dependency can be proved sound by observing that the initial color of variables can be designated by the name of these variables and that the color mix at point ℓ for variable y is $\{\mathsf{x} \mid \mathcal{S}^{+\infty}[\![\mathsf{P}]\!] \in \mathcal{D}^\ell\langle \mathsf{x}, \mathsf{y}\rangle\}$. Note that in the postulated instrumented semantics, the choice of diff remains implicit as defined by the arbitrarily selected color mixing rules. Otherwise the instrumented semantics [38] need to be semantically justified with respect to the non-instrumented semantics, in which case the non-instrumented semantics can be used as well to justify dependency, as we do.

Tracking analysis Assume the initial values of variables (more generally inputs) are partitioned into tracked $\mathbb{T}$ and untracked $\mathbb{U}$ variables, $\mathbb{V} = \mathbb{T} \cup \mathbb{U}$ and $\mathbb{T} \cap \mathbb{U} = \varnothing$. The tracking abstraction $\alpha^\tau(\mathbf{D})$ of a dependency property $\mathbf{D} \in \mathbb{L} \to \wp(\mathbb{V} \times \mathbb{V})$ (7) attaches to each program point ℓ the set of variables y which, at that program point ℓ, depend upon the initial value of at least one tracked variable $\mathsf{x} \in \mathbb{T}$.

$$\alpha^\tau(\mathbf{D})\ell \triangleq \{\mathsf{y} \mid \exists \mathsf{x} \in \mathbb{T} \,.\, \langle \mathsf{x}, \mathsf{y}\rangle \in \mathbf{D}(\ell)\}$$

A tracking analysis is an over-approximation of the abstract tracking semantics

$$\mathcal{S}^\tau[\![\mathsf{S}]\!] \supseteq \alpha^\tau(\alpha^{\mathrm{d}}(\{\mathcal{S}^{+\infty}[\![\mathsf{S}]\!]\}))$$

assigning the each program point ℓ, a set $\mathcal{S}^\tau[\![\mathsf{S}]\!]\ell \in \wp(\mathbb{V})$ of variables potentially depending on tracked variables. Examples are taint analysis in privacy/security checks [28,61] (tracked is tainted, untracked is untainted); binding time analysis in offline partial evaluation [33] (tracked is dynamic, untracked is static) and absence of interference [30,66,15,36,45] (tracked is high (private/untrusted), untracked is low (public/trusted)).

8 Conclusion

Related work Definitions of dependency follow one of the approaches below[1].

[1] Some approaches are a mix of these cases. For example [69,19] postulates dependency on one trace as in 1. and then abstracts for a set of traces as in 3. and so uses the "Merge over all paths" approach of dataflow analysis [23], with no semantics justification of soundness.

1. Dependency is postulated for a given programming language by specifying an algorithm [68,26] or a calculus [1] which is claimed, a priori, to define dependency. Since the definition is not semantic, it hides (and does not allow to discuss) important details and so is hardly transferable to other languages.
2. Dependency is incorporated in a semantics of the language instrumented with a policy [65] or flows [17]. The problem is that changing slightly the instrumentation definitely changes the variety of dependency which is defined. In particular, it does not guarantee that the notion of dependency is defined uniformly all over the language (*e.g.* conditionals and iterations might be handled using different notions of dependency).
3. Dependency is defined as an abstract interpretation of properties of a formal semantics [65,8,64], although the abstraction originally remained completely implicit [30,66].

Our approach is in the category 3. Besides a generalization beyond input-output dependency, we have shown that, although dependency is an "hyperproperty" (*i.e.* a property of the semantics which is a set of traces), we don't need a different abstract interpretation theory for that case (as in [8,64] introducing specific collecting semantics abstracting general semantic properties). The classical approach [21,23] directly applies whichever kind of property is considered.

Achievements We have designed by calculus a new potential value dependency analysis between the initial value of variables and their value when reaching a program point during execution. It follows and formalizes the intuition provided by [26], "Information flows from object $\mathtt{x}$ to object $\mathtt{y}$, denoted $\mathtt{x} \rightsquigarrow \mathtt{y}$, whenever information stored in $\mathtt{x}$ is transferred to, or used to derive information transferred to, object $\mathtt{y}$. A program statement specifies a flow $\mathtt{x} \rightsquigarrow \mathtt{y}$ if execution of the statement could result in a flow $\mathtt{x} \rightsquigarrow \mathtt{y}$." "Information flow" is formalized as "changing initial values will change the non-empty sequence of values observed at a program point".

An alternative [32,13,35] is to monitor an abstraction of the program semantics at runtime (Lem. 1 on prefix observation is not valid for all definitions of dependency so dynamic checking might be unsound [60,9]).

The analysis is not postulated but derived formally by abstract interpretation of the trace semantics. So our definition is concise and coherent. We found no need for extra notions like (hyper)^{n}properties [8], non-standard abstract interpretation [64], postulated instrumented semantics [70, Sect. 4], multisemantics [16], monadic reification [31], *etc.*

As shown by [13,34] and Ex. 7, taking values into account will definitely improve the precision of the dependency analysis. As noticed in Section **6**, one possible implementation is by a reduced product of a dependency analysis with a reachability analysis [69,19].

The data-dependence analysis used to detect parallelism in sequential code [54] is also an abstract interpretation, see [63].

Future work The Def. 1 of $\mathcal{D}$ is certainly not unique. For example replacing diff in (3) by equality would take into consideration timing dependencies (Ex. 5) and empty observations (Ex. 6). The methodology that we proposed in this paper can, in our opinion, be applied to a wide variety of definitions of dependency, as follows.

The semantics is a set of executions by pairs $\langle \pi^{\ell}, {}^{\ell}\pi' \rangle$ where π^{ℓ} is the past before reaching ℓ and ${}^{\ell}\pi'$ is the continuation. We define an abstraction of the past π^{ℓ} (*e.g.* the initial value of variables in our case). We define an abstraction of the continuation (*e.g.* seqval (1) in our case). We define the difference between past abstractions (the initial values of variables only differ for one variable in our case). We define the difference between futures (diff in our case). Then the abstraction of the future depends on the abstraction of the past if and only if there exist two executions with different past abstractions and different future abstractions. We conjecture that by varying the past/future abstractions and their difference, we can express the dependency abstractions introduced in the literature. Good examples are [29] for (abstract) non-interference and [12] for mitigation against side-channel attacks.

Acknowledgement. I thank the reviewers for their comments. This work was supported in part by NSF Grant CCF-1617717. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: POPL. pp. 147–160. ACM (1999)
2. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models (extended version). Formal Methods in System Design **40**(2), 170–205 (2012)
3. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing **2**(3), 117–126 (1987)
4. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL. pp. 91–102. ACM (2006)
5. Amtoft, T., Banerjee, A.: A logic for information flow analysis with an application to forward slicing of simple imperative programs. Sci. Comput. Program. **64**(1), 3–28 (2007)
6. Andrews, G.R., Reitman, R.P.: An axiomatic approach to information flow in programs. ACM Trans. Program. Lang. Syst. **2**(1), 56–76 (1980)
7. Apel, S., Kästner, C., Batory, D.S.: Program refactoring using functional aspects. In: GPCE. pp. 161–170. ACM (2008)
8. Assaf, M., Naumann, D.A., Signoles, J., Éric Totel, Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: POPL. pp. 874–887. ACM (2017)
9. Balliu, M., Schoepe, D., Sabelfeld, A.: We are family: Relating information-flow trackers. In: ESORICS (1). Lecture Notes in Computer Science, vol. 10492, pp. 124–145. Springer (2017)
10. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. Lecture Notes in Computer Science, vol. 6664, pp. 200–214. Springer (2011)

11. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011)
12. Barthe, G., Grégoire, B., Laporte, V.: Provably secure compilation of side-channel countermeasures. IACR Cryptology ePrint Archive **2017**, 1233 (2017)
13. Bello, L., Hedin, D., Sabelfeld, A.: Value sensitivity and observable abstract values for information flow control. In: LPAR. Lecture Notes in Computer Science, vol. 9450, pp. 63–78. Springer (2015)
14. Bergeretti, J., Carré, B.: Information-flow and data-flow analysis of while-programs. ACM Trans. Program. Lang. Syst. **7**(1), 37–61 (1985)
15. Bowman, W.J., Ahmed, A.: Noninterference for free. In: ICFP. pp. 101–113. ACM (2015)
16. Cabon, G., Schmitt, A.: Annotated multisemantics to prove non-interference analyses. In: PLAS@CCS. pp. 49–62. ACM (2017)
17. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. Mathematical Structures in Computer Science **21**(6), 1301–1337 (2011)
18. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010)
19. Cortesi, A., Ferrara, P., Halder, R., Zanioli, M.: Combining symbolic and numerical domains for information leakage analysis. Trans. Computational Science **31**, 98–135 (2018)
20. Cousot, P.: Syntactic and semantic soundness of structural dataflow analysis. In: SAS. Lecture Notes in Computer Science, vol. this volume. Springer (2019)
21. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
22. Cousot, P., Cousot, R.: Constructive versions of Tarski's fixed point theorems. Pacific Journal of Mathematics **81**(1), 43–57 (1979)
23. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282. ACM Press (1979)
24. Cousot, P., Cousot, R., Mauborgne, L.: Theories, solvers and static analysis by abstract interpretation. J. ACM **59**(6), 31:1–31:56 (2012)
25. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does Astrée scale up? Formal Methods in System Design **35**(3), 229–264 (2009)
26. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
27. Fagin, R., Vardi, M.Y.: The theory of data dependencies - a survey. In: Mathematics of Information Processing. Proceedings of Symposia in Applied Mathematics, vol. 34, pp. 19–71. AMS (1986)
28. Ferrara, P., Olivieri, L., Spoto, F.: Tailoring taint analysis to GDPR. In: Privacy Technologies and Policy. 6th Annual Privacy Forum, APF 2018, Barcelona, Spain, June 13-14, 2018, Revised Selected Papers (Jun 2018). https://doi.org/10.1007/978-3-030-02547-2_4
29. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: A unifying framework for weakening information-flow. ACM Trans. Priv. Secur. **21**(2), 9:1–9:31 (2018)
30. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: IEEE Symposium on Security and Privacy. pp. 75–87. IEEE Computer Society (1984)
31. Grimm, N., Maillard, K., Fournet, C., Hritcu, C., Maffei, M., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N., Béguelin, S.Z.: A monadic framework for relational verification: applied to information security, program equivalence, and optimizations. In: CPP. pp. 130–145. ACM (2018)

32. Guernic, G.L.: Confidentiality Enforcement Using Dynamic Information Flow Analyses. Ph.D. thesis, Kansas State University, United States of America (2007)
33. Hatcliff, J.: An introduction to online and offline partial evaluation using a simple flowchart language. In: Partial Evaluation. Lecture Notes in Computer Science, vol. 1706, pp. 20–82. Springer (1998)
34. Hedin, D., Bello, L., Sabelfeld, A.: Value-sensitive hybrid information flow control for a javascript-like language. In: CSF. pp. 351–365. IEEE Computer Society (2015)
35. Hedin, D., Bello, L., Sabelfeld, A.: Information-flow security for javascript and its apis. Journal of Computer Security **24**(2), 181–234 (2016)
36. Heinze, T.S., Turker, J.: Certified information flow analysis of service implementations. In: SOCA. pp. 177–184. IEEE Computer Society (2018)
37. Hong, H.S., Lee, I., Sokolsky, O.: Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In: SCAM. pp. 25–34. IEEE Computer Society (2005)
38. Jones, N.D., Nielson, F.: Abstract interpretation: a semantics-based tool for program analysis. In: Abramsky, S., Gabbay, D.M. (eds.) Handbook of Logic in Computer Science, vol. 4, Semantic Modelling, pp. 527–636. Oxford University Press (1995)
39. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL. pp. 247–259. ACM (2015)
40. Karr, M.: Affine relationships among variables of a program. Acta Inf. **6**, 133–151 (1976)
41. Kildall, G.A.: A unified approach to global program optimization. In: POPL. pp. 194–206. ACM Press (1973)
42. Knoop, J., Rüthing, O.: Constant propagation on the value graph: Simple constants and beyond. In: CC. Lecture Notes in Computer Science, vol. 1781, pp. 94–109. Springer (2000)
43. Kranjc, A.: Tracer Hydrology 97. CRC Press (Jan 1997)
44. Lampson, B.W.: A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)
45. Lourenço, L., Caires, L.: Dependent information flow types. In: POPL. pp. 317–328. ACM (2015)
46. Malburg, J., Finder, A., Fey, G.: Debugging hardware designs using dynamic dependency graphs. Microprocessors and Microsystems - Embedded Hardware Design **47**, 347–359 (2016)
47. Mandal, A.K., Cortesi, A., Ferrara, P., Panarotto, F., Spoto, F.: Vulnerability analysis of Android auto infotainment apps. In: CF. pp. 183–190. ACM (2018)
48. Mantel, H.: A Uniform Framework for the Formal Specification and Verification of Information Flow Security. Dr.-ing. thesis, Fakultät I der Universität des Saarlandes, Saarbrücken, Germany (Jul 2003)
49. Mastroeni, I., Zanardini, D.: Abstract program slicing: An abstract interpretation-based approach to program slicing. ACM Trans. Comput. Log. **18**(1), 7:1–7:58 (2017)
50. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006)
51. Muthukumar, K., Hermenegildo, M.V.: Compile-time derivation of variable dependency using abstract interpretation. J. Log. Program. **13**(2&3), 315–347 (1992)
52. Müller-Olm, M., Rüthing, O.: On the complexity of constant propagation. In: ESOP. Lecture Notes in Computer Science, vol. 2028, pp. 190–205. Springer (2001)
53. Ngo, M., Naumann, D.A., Rezk, T.: Typed-based relaxed noninterference for free. CoRR **abs/1905.00922** (2019)

54. Padua, D.A., Wolfe, M.: Advanced compiler optimizations for supercomputers. Commun. ACM **29**(12), 1184–1201 (1986)
55. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. **74**(1), 358–366 (1953)
56. Rival, X.: Abstract dependences for alarm diagnosis. In: APLAS. Lecture Notes in Computer Science, vol. 3780, pp. 347–363. Springer (2005)
57. Rival, X.: Understanding the origin of alarms in astrée. In: SAS. Lecture Notes in Computer Science, vol. 3672, pp. 303–319. Springer (2005)
58. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
59. Sadeghi, A., Bagheri, H., Garcia, J., Malek, S.: A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. IEEE Trans. Software Eng. **43**(6), 492–530 (2017)
60. Schoepe, D., Balliu, M., Pierce, B.C., Sabelfeld, A.: Explicit secrecy: A policy for taint tracking. In: EuroS&P. pp. 15–30. IEEE (2016)
61. Spoto, F., Burato, E., Ernst, M.D., Ferrara, P., Lovato, A., Macedonio, D., Spiridon, C.: Static identification of injection attacks in java. ACM Trans. Program. Lang. Syst. **41**(3), 18:1–18:58 (2019)
62. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific J. of Math. **5**, 285–310 (1955)
63. Tzolovski, S.: Data dependence as abstract interpretations. In: SAS. Lecture Notes in Computer Science, vol. 1302, p. 366. Springer (1997)
64. Urban, C., Müller, P.: An abstract interpretation framework for input data usage. In: ESOP. Lecture Notes in Computer Science, vol. 10801, pp. 683–710. Springer (2018)
65. Volpano, D.M.: Safety versus secrecy. In: SAS. Lecture Notes in Computer Science, vol. 1694, pp. 303–311. Springer (1999)
66. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2/3), 167–188 (1996)
67. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. **13**(2), 181–210 (1991)
68. Weiser, M.: Program slicing. IEEE Trans. Software Eng. **10**(4), 352–357 (1984)
69. Zanioli, M., Cortesi, A.: Information leakage analysis by abstract interpretation. In: SOFSEM. Lecture Notes in Computer Science, vol. 6543, pp. 545–557. Springer (2011)
70. Ørbæk, P.: Can you trust your data? In: TAPSOFT. Lecture Notes in Computer Science, vol. 915, pp. 575–589. Springer (1995)