Data Dependencies and Program Slicing: from Syntax to Abstract Semantics

Isabella Mastroeni
Università di Verona
isabella.mastroeni@univr.it

Damiano Zanardini
Universidad Politécnica de Madrid
damiano@clip.dia.fi.upm.es

Abstract

We discuss the relation between program slicing and data dependencies. We claim that slicing can be defined, and therefore calculated, parametrically on the chosen notion of dependency, which implies a different result when building the program dependency graph. In this framework, it is possible to choose dependency in the syntactic or semantic sense, thus leading to compute possibly different, smaller slices. Moreover, the notion of abstract dependency, based on properties instead of exact data values, is investigated in its theoretical meaning. Constructive ideas are given to compute abstract dependencies on expressions, and to transform properties in order to rule out some dependencies. The application of these ideas to information flow is also discussed.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal Methods; D.3.1 [*Formal Definitions and Theory*]: Semantics; F.3.2 [*Semantics of Programming Languages*]: Program analysis; I.1.0 [*General*]

General Terms Languages, Security, Theory, Verification

Keywords Abstract Interpretation, Abstract non-interference, Dependency analysis, Program slicing

1. Introduction

Control and data-flow analysis is among the most effective techniques for program understanding, verification and debugging. Efficient algorithms for intra and inter-procedural program manipulation have been designed on suitable program representations called *program dependency graphs* (denoted PDGs), which keep track of how information propagates through the program code. This is the case of (static) *program slicing* [22, 17, 16, 20, 4], a technique which extracts from programs the statements which are *relevant* to a given behavior. In particular, a *slice* is an executable program which is obtained by statement deletion on the original program, and whose behavior must be identical to a specific subset of the original behavior. Informally, a slice has to be such that an *observer* should not distinguish between the execution of the program and the slice, if (s)he only pays attention to the value of a set of variables at some program point, as specified in the *slicing criterion*.

Since the first publications [22], there have been many works proposing several notions of slicing, and different algorithms to compute slices. Ward and Zedan [21] note that most papers provide an informal definition of the meaning of a program slice, and concentrate attention rather on defining and computing program dependencies. According to the authors, this focus on the computation of program dependencies somehow confuses the notion of slice with the algorithms for computing slices. Their work looks at slicing as a program transformation, and does not even rely on data dependencies.

Main contribution

The construction of a program dependency graph relies on the chosen notion of dependency. We believe it is possible to define a general notion of slicing, parametrically on what the words *depends on* (or the symmetric ones *is relevant to*) mean. The idea is that we choose the dependency, and derive a corresponding notion of slicing.

In literature, characterizing slicing by means of dependency is not new. In particular, Amtoft and Banerjee [3] exploit a logic for deriving independencies in order to certify if a program transformation is a slice of the original program. In that work, as well as in most approaches to slicing (e.g., the algorithm by Reps [16], based on PDGs), slicing is characterized by means of a syntactic dependency, based on the occurrence of a variable in an expression. For instance, in the assignment $x := 2y$, we can state that x depends on y, because y occurs in the assigned expression. This notion of dependency loses some information, when syntactic occurrence is not enough to get the real idea of relevancy. For instance, the value assigned to x does not depend on y in the statement $x := z + y - y$, although y occurs in the expression. The syntactic approach may fail in computing the optimal set of dependencies, since it is not able to rule out this kind of false dependencies. This results in obtaining a slice which contains more statements than needed.

On the other hand, moving from standard syntactic slicing to a semantic-based slicing, where x is the only variable relevant to $x + y - y$, would result in more precise slices, only considering *true* dependencies and their propagation.

In the same framework, dependencies can be *abstracted*, in order to obtain a weaker notion which only holds when some *property* of data depends on a property of some variable. More precisely, x depends on y only if some property of x is affected by some property of y at a previous point in the execution. This leads to a weaker notion of dependency, called *abstract dependency*. The main interest is in deciding, in the evaluation of an expression e, which (properties of) data the final abstract value of e depends on, i.e., which variables are relevant to the chosen property of the expression e. Computing slices on abstract dependencies gives a smaller program, since the abstraction *prunes* the PDG by ruling out some dependencies which are only relevant at the *concrete* level.

The present paper proposes a definition of abstract dependency, based on the *concrete semantics* of the program (see Sec. 4). This notion of dependency is parametric on the properties of interest. Basically, an expression e depends on a variable x w.r.t. a property ρ if changing x, and keeping all other variables unchanged with respect to ρ, may lead to a change in e with respect to ρ (here, we take the same property for both the input variables and the output value of the expression but, in principle, they might be different properties). Since this notion is inherently semantic, the problem arises of making it implementable. A constructive approach is introduced, which:

- computes the variables an expression depends on, given some properties on data;
- finds an approximation of the most precise property s.t. an expression does not depend on a chosen variable.

So far, only denotational characterizations of abstract data dependencies have been proposed. Our purpose is to partially fill the gap between semantics and computation, by giving a constructive, approximated way for computing dependencies.

Finally, an application to information flow, particularly to abstract non-interference (ANI) [9], is discussed. Light is shed on the relation between ANI and abstract dependencies, by pointing out how computing dependency may be used in order to enforce confidentiality of data. To this purpose, and according to the definition of ANI, it makes sense to define another version of abstract dependencies, which is based on the abstract, rather than the concrete semantics. Both notions can be possibly used in information flow. In particular, the latter one can rule out some *false* dependencies, which are actually generated by harmless flows of information. On the other hand, it relies on some assumptions about how the attacker knows the code of the program and can statically analyze it [9], namely, on its precision in analyzing it. A constructive approach is also introduced for this second version of dependency.

State of the art and related work

Besides the foundational work by Cohen [6] and Cartwright and Felleisen [5], a (standard) dependency calculus was proposed by Abadi et al. [1] in the setting of functional languages. More recently, Amtoft and Banerjee [3] defined a Hoare-style logic to analyze variable *independency*; program traces, potentially infinitely many, are abstracted, in the framework of abstract interpretation, by a finite set of variable independencies. The potentiality of this approach, which will be introduced more deeply afterwards, is that independencies can be statically checked against the logic and applied, as shown by the authors, to program slicing. This work was extended to object-oriented languages [2], and the independency analysis has been provided.

Rival [18] recently characterized abstract dependencies. This is, to the best of our knowledge, the only description of a weakened form of dependency, based on representing data properties by means of abstract interpretation. Rival discusses abstract dependency and its application to alarm diagnosis, together with several techniques for analyzing and composing dependencies (e.g., the dependency analysis of the compound statement $[s_1 \ ; \ s_2]$ involves combining dependencies in s_1 and s_2 compositionally). However, for non-compound statements, only a mathematical, set-theoretic definition of dependencies is provided; the purpose of our constructive approach is precisely to shed light on how the calculus of abstract dependency may be done on statements.

Literature on program slicing is quite extended, and almost spans the last three decades (see Tip [20] for a survey). An abstract version (often referred to as *abstract slicing*), based on data properties instead of exact values, still lacks a comprehensive definition. Despite its title, the work by Hong, Lee and Sokolsky [13] discusses a different notion of abstract slicing, where abstract interpretation is considered in the restricted area of predicate abstraction and where, instead of weakening the observation of all the executions, the authors only look at a subset of the possible executions.

2. Foundations

This section, which is not supposed to be exhaustive, provides the necessary background in the Abstract Interpretation theory, and introduces the simple programming language the paper refers to. Moreover, further notions on partitioning abstract domains are given, which will be important in motivating an assumption underlying the rest of the paper (Sec. 4).

Abstract interpretation: a brief introduction

In the following, we will use the standard framework of *abstract interpretation* [7, 8] for modeling data properties. Abstract domains are chosen for denoting properties over concrete domains, since their mathematical structure guarantees, for each concrete element, the existence of the *best correct approximation* in the abstract domain. This is due to the property of abstract domains of being closed under greatest lower bound. Formally, the *lattice of abstract interpretations of* $\mathcal{C}$ is isomorphic to the lattice $\text{UCO}(\mathcal{C})$ of all the *upper closure operators* (uco) on $\mathcal{C}$ [8]. An uco $\rho : \mathcal{C} \mapsto \mathcal{C}$ on a poset $\mathcal{C}$ is monotone, idempotent, and extensive[1]. Ucos are uniquely determined by the set $\rho(\mathcal{C})$ of their fix-points (we will abuse notation by identifying $\rho(\mathcal{C}) = \{\rho(c) | c \in \mathcal{C}\}$ with ρ). For a singleton $\{v\}$ and a uco ρ, we often write $\rho(v)$ instead of $\rho(\{v\})$. In the following, we will use the abstract domain SIGN, containing $[\top]$, $[\bot]$ and the abstract values $[\mathbf{neg}] \equiv \mathbb{Z}^-$ (negative numbers) and $[\mathbf{pos}] \equiv \mathbb{Z}^+$ (positive numbers with 0). The corresponding uco maps sets of numbers to their sign. Moreover, $\text{PAR} = \{[\top], [\mathbf{even}], [\mathbf{odd}], [\bot]\}$ models parity of numbers. Finally, $\text{PARSIGN} = \text{PAR} \sqcap \text{SIGN}$ is obtained by reduced product, i.e., it is the smallest domain which is more precise than both PAR and SIGN (e.g., it contains $[\mathbf{poseven}]$).

Completeness in abstract interpretation is a property of abstract domains relative to a fixed computation. An abstract domain ρ is complete for f if it is optimally precise for the computation of f. Formally, ρ is complete for f if $\rho \circ f \circ \rho = \rho \circ f$. In other words, computing f in the abstract domain corresponds precisely

[1] $\forall X \in \mathcal{C}.\ X \leq_{\mathcal{C}} \rho(X)$. Usually, $\mathcal{C} = \wp(\mathcal{D})$ for some $\mathcal{D}$, and $\leq_{\mathcal{C}}$ is set inclusion on $\mathcal{D}$.

to abstracting the concrete computation of f, without further loss of information. E.g., PAR is complete for $+$, but SIGN is not.

Language and Semantics

We consider the IMP language [23] and denote by $\mathbb{V} \overset{\text{def}}{=} \mathbb{Z}$ the set of *values* for the static variables VAR. Expressions $e \in$ EXP are defined by standard operators on constants and variables. *States* $\sigma \in \Sigma = \text{VAR} \rightarrow \mathbb{V}$ are memory configurations. i.e., mappings from variables to values. In general, if a program has k variables $x_1, \ldots, x_k$, we will represent states as tuples, i.e., $\Sigma = \mathbb{V}^k$ and $\sigma = \langle v_1, \ldots, v_k \rangle$; the state $\sigma' = \sigma[x \leftarrow v]$ satisfies $\sigma'(x) = v$ and $\sigma'(y) = \sigma(y)$ for any $y \neq x$.

As far as semantics is concerned, $\mathcal{S}[\![s]\!](\sigma)$ is the state obtained by computing s in σ. Moreover, $\mathcal{P}[\![s]\!]_p(\sigma)$ is a *partial semantics* collecting all the possible states at the program point p inside s, when s is executed in σ. [2] We write $e[x_1, \ldots, x_k]$ to make explicit the variables in e (i.e., $\text{VARS}(e) = \{x_1, \ldots, x_k\}$); $\mathcal{E}[\![e]\!](\sigma)$ is the (concrete) semantics of expressions. Consider now an abstract domain ρ on values: the related *abstract semantics* on expressions, $\mathcal{E}[\![e]\!]^\rho$, is applied to abstract states $\Sigma^\rho \overset{\text{def}}{=} \rho(\wp(\mathbb{V}))^k$ and is defined as the *best correct approximation* of $\mathcal{E}[\![e]\!]$. Namely, let $\sigma = \langle v_1, \ldots, v_k \rangle \in \Sigma$ and $\varepsilon = \langle \rho(v_1), \ldots, \rho(v_k) \rangle \in \Sigma^\rho$: $\mathcal{E}[\![e]\!]^\rho(\varepsilon) = \rho(\{\mathcal{E}[\![e]\!](u_1, \ldots, u_k) \mid \forall i.\ u_i \in \rho(v_i)\})$.

Similarly, semantics $\mathcal{P}[\![s]\!]^\rho_p(\varepsilon_0)$ (where $\varepsilon_0 = \langle \top, \ldots, \top \rangle$ is the most abstract state, mapping all variables to $\top$) statically computes a safe over-approximation of the minimal abstract state $\varepsilon'_p \in \Sigma^\rho$ which describes variables at p:

$$\mathcal{P}[\![s]\!]^\rho_p(\varepsilon_0) \overset{\text{def}}{=} \varepsilon_p \geq \varepsilon'_p = \rho(\cup\{\mathcal{P}[\![s]\!]_p(\sigma) \,|\sigma \in \Sigma\}).$$

When e is clear by the context, $\forall x$ is a shorthand for $\forall x \in \text{VARS}(e)$. Ordering $\varepsilon' \leq \varepsilon''$ and abstraction $\rho(\sigma)$ are defined pointwise. Given a state ε, a *covering* $\{\varepsilon_1..\varepsilon_k\}$ is a set of states such that ε describes the same set of concrete states as all the ε_i: $\varepsilon = \cup_i \varepsilon_i$. $\forall \varepsilon$ will stand for $\forall \varepsilon \leq \varepsilon_p$.

Partitions and atoms

Given $\rho \in \text{UCO}(\wp(\mathbb{V}))$, the *induced partition* $\Pi(\rho)$ of ρ is the set $\{V_1, .., V_k\}$, partition of $\mathbb{V}$, characterizing classes of values undistinguished by ρ: $\forall i. \forall x, y \in V_i\ .\rho(x) = \rho(y)$.

EXAMPLE 2.1. *Let* $\rho = \{[\top], [\mathbf{even}]\}$*; the induced partition* $\Pi(\rho)$ *is* $\{[\mathbf{even}], [\mathbf{odd}]\}$ *since values are distinguished by their parity.* $\Pi(\rho)$ *happens to be equal to* $\Pi(\text{PAR})$*, i.e., they induce the same partition though* PAR *is more concrete.*

A domain ρ is *partitioning* if it is the most concrete among those inducing the same partition: for a partition P, $\rho = \sqcap\{\eta | \Pi(\eta) = P\}$. Every η can be transformed into ρ' partitioning by closing it under set complement of abstract values (in the example, $[\mathbf{odd}]$ is the complement of $[\mathbf{even}]$ w.r.t. $[\top]$) [15]. If ρ is partitioning, $\Pi(\rho)$ is the set of the atoms of ρ, viewed as a complete lattice[3], i.e., the atoms of a partitioning domain are the abstractions of singletons. $\text{ATOM}(V)$ is the atomicity predicate and can be extended pointwise to states.

[2] The values can be more than one. E.g., in **while** b **do** ... ; $y := y + 1$ **od**, the set of values y can have *inside* the loop grows at each iteration, while, at the exit, y has a unique value resulting from the termination of the loop (if it terminates).

[3] $V \in \rho \smallsetminus \{\bot\}$ is an *atom* iff $\bot \leq_\rho U \leq_\rho V$ implies $U = V$ or $U = \bot$ for every $U \in \rho$

3. Slicing vs. dependencies

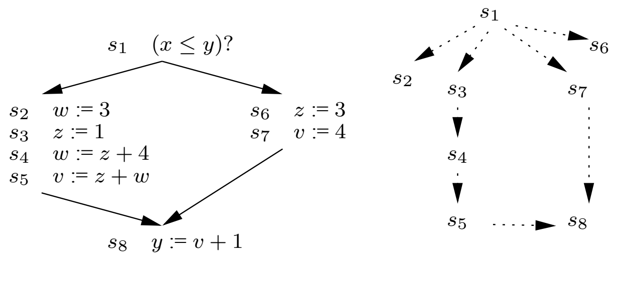
Program slicing [22] is a program manipulation technique which extracts, from programs, statements relevant to a particular computation. Informally, a slice provides the answer to the question: *which program statements potentially affect the computation of the variable x at the program point (i.e., statement) s?* In order to answer this question, an observer needs a *window* through which only part of the program states can be seen [4]. Therefore, a *program slice* is the set of program statements which contribute directly or indirectly to the values assumed by some set of variables at some program point. The observation window of the program is specified by a *slicing criterion*, usually represented as a pair $\langle s, X \rangle$, which requires the observation of the variables X at the program points s. The following definition [4] formalizes the original idea of program slicing by Weiser [22]:

DEFINITION 3.1. *For a statement (program point) s and a variable x, the (static backward executable) slice* P' *of the program P with respect to the slicing criterion* $\langle s, \{x\} \rangle$ *is any executable program with the following properties:*

1. P' *can be obtained by deleting zero or more statements from P;*
2. *If P halts on the input I, then, each time s is reached in P, the value of x at s, is the same in P and in* P'*. If P fails to terminate, then s may be reached more times in* P' *than in P, but P and* P' *have the same value for x each time s is executed by P.*

The standard approach to characterizing slices, and the corresponding relation *being slice of*, are based on the notion of program dependency graph [14, 16], as described by Binkley and Gallagher [4]. *Dependency graphs* can be built out of programs, and describe how data propagate at runtime. Following the program slicing approach, we could be interested in computing dependencies on statements: s'' depends on s' if some variables which are used inside s'' are defined inside s', and definitions in s' reach s'' through at least one possible execution path. Also, s depends *implicitly* on an if-statement or a loop if its execution depends on the boolean guard.

EXAMPLE 3.2. *Consider the program below and the derived dependency graph (edges which can be obtained by transitivity are omitted):* s_8 *depends on both* s_5 *and* s_7 *(and, by transitivity,* s_1*)*



since v is not known statically when entering s_8*. On the other hand, there is* no *dependency of* s_8 *on either (i)* s_6*, since z is not used in* s_8*; or (ii)* s_2*, since w is always redefined before* s_8*. The dependency of* s_7 *on* s_1 *is implicit since* 4 *does not depend on x nor y, but* s_7 *is executed conditionally on* s_1*.*

There exist a number of techniques for building and analyzing dependency graphs, allowing to study how information propagates among statements. Usually, the basic rules for detecting a dependency between s_1 and s_2 are

- *Control dependence edges:* s_1 represents a control predicate and s_2 represents a component of the program immediately nested withing the control predicate s_1;
- *Flow dependence edges:* s_1 defines a variable x which is used in s_2, i.e., $x \in \mathbf{def}\,(s_1) \cap \mathbf{ref}\,(s_2)$ and x is not further defined in any statement between s_1 and s_2.

As noted by Ward [21], there is clearly a *gap* between the definition of slicing, given in Def. 3.1, and the standard implementation based on program dependency graphs, outlined above. Anyway, we think that this is not because, in general, the notion of dependency is something untied to slicing and simply used for implementing. Rather, because slicing and dependencies are usually defined at *different levels* of approximation. In particular, we can note that Def. 3.1 defines slicing by requiring the same *behavior*, w.r.t. a criterion, between the program and the slice, i.e., we are specifying what is *relevant* as a *semantic* requirement. On the other hand, we can see above that PDGs consider a notion of dependency between statements which corresponds to the *syntactic* presence of a variable in the definition of another variable. In other words, slices are usually defined at the *semantic* level, while dependencies are used for implementation and therefore defined at the *syntactic* level. This means that, the gap between slicing and dependencies is due to the well known gap between semantics and syntax. In this paper, the idea is to partially fill this gap by identifying a notion of *semantic* dependency corresponding to the slicing definition given above, in order to characterize the implicit parametricity of the notion of slicing on a corresponding notion of dependency. Note that the PDG approach is based on the computation of *used* variables in the expression e, i.e., the variables e depends on. Therefore, we wonder if this set can be rewritten by considering a semantic form of dependency leading towards standard slicing. Moreover, we study how this dependency can be then replaced by other forms of dependencies, modeling what is *relevant to a given computation*, and this results in generating different weakenings of slicing.

One of the first works aiming to formalize the notion of dependency is the information flow *logic* by Amtoft and Banerjee [3]. This logic allows to formally derive, by structural induction, the set of all the *independencies* among variables. In Fig. 1, we use the original notation proposed by the authors, where $[x \ltimes y]$ is read as *the current value of x is independent of the initial value of y*, and holds if, for each pair of *initial* states which agree on all the variables but y, the corresponding *current* states agree on x. Hence, $T^{\#}$ stands for sets of independencies, and G is a set of variables representing the *context*, i.e., (a superset of) the variables at least one test surrounding the statements depends on. Moreover, in the rules we introduce a new notation which will be useful in the following: $y \rightsquigarrow e \;\Leftrightarrow\; y \in \textsc{vars}\,(e)$.

In our aim of defining slicing in terms of dependencies, the first thing we have to observe on this logic is that it always computes (in)dependencies from the *initial* values of variables. This fact, makes its use for slicing not so straightforward, since it loses the *local* dependency between statements. In order to understand what we mean, consider for example the program fragment $P = w := x+1; y := w+2; z := y+3$. At the end of this program, we know that z only depends on the initial value of x, but, by using the

$$\begin{array}{l}
G \vdash \{T_0^{\#}\}\; x := e\; \{T^{\#}\} \\
\qquad \text{if}\, \forall [y \ltimes w] \in T^{\#}.\, (x \neq y \;\Rightarrow\; [y \ltimes w] \in T_0^{\#}) \\
\qquad (x = y \;\Rightarrow\; (w \notin G \wedge \forall z \rightsquigarrow e.\, [z \ltimes w] \in T_0^{\#})
\end{array}$$

$$\frac{G_0 \vdash \{T_0^{\#}\} s_1 \{T^{\#}\} \;\; G_0 \vdash \{T_0^{\#}\} s_2 \{T^{\#}\}}{G \vdash \{T_0^{\#}\} \textbf{if}\; e\; \textbf{then}\; s_1\; \textbf{else}\; s_2 \{T^{\#}\}}$$
$$\qquad \text{if}\, G \subseteq G_0 \;\wedge\; (w \notin G_0 \;\Rightarrow\; \forall x \rightsquigarrow e.\, [x \ltimes w] \in T_0^{\#})$$

$$\frac{G_0 \vdash \{T^{\#}\} s \{T^{\#}\}}{G \vdash \{T^{\#}\} \textbf{while}\; e\; \textbf{do}\; s \{T^{\#}\}}$$
$$\qquad \text{if}\, G \subseteq G_0 \;\wedge\; (w \notin G_0 \;\Rightarrow\; \forall x \rightsquigarrow e.\, [x \ltimes w] \in T^{\#})$$

Figure 1. A fragment of the independency logic

logic in Fig. 1, we lose the trace of (in)dependencies which, in this case, would involve all the three assignments. Indeed, this logic, is more suitable for forward slicing (which is the one considered by the authors [3]), since it fixes the criterion on the input. In fact, forward slicing finds out all the statements affected by the variables in the criterion. In the trivial example given above, if we consider as criterion the input of x, then we obtain that all the statements depend on x. Therefore, any slice of the original program contains all of them [3].

Now, let us make some observations. In this section we described two different approaches to slicing, one based on PDGs and the other based on a logic for independencies. What we would like to underline is that even if these methods follow opposite directions from the criterion for computing slices, both are explicitly and strongly based on a notion of dependency which is clearly syntactic. In PDGs, the notion of dependency is implicit in the definition of the set $\mathbf{ref}\,(s)$, where the set of all the variables referred in the evaluation of the expression e in s is exactly the set of all the variables appearing in the expression e, i.e., all the variables x such that $x \rightsquigarrow e$. In the logic, more explicitly, we use exactly this notion of dependency for characterizing the set of independencies holding during the execution of a program.

These observations lead us to the possibility of defining slicing in terms of dependencies by forgetting the way we want to compute it, for instance in the forward or in the backward direction. At this point, in order to obtain different notions of slicing we can simply focus our attention on how we can define new kind of dependencies. The first step towards a generalization of the way of defining slicing is to consider *semantic dependencies*, where intuitively a variable is relevant for an expression if it is relevant for its *evaluation*. For example, $z + y - y$ does not semantically depend on y. This semantic notion can then easily generalized in what we will call *abstract dependency*, where a variable is relevant to an expression if it affects a given *property* of its evaluation.

4. Abstract dependencies for program slicing

As explained in Sec. 3, we are particularly interested in finding which variables might affect the evaluation of e in the assignment $x := e$ or in a control statement with guard e, i.e., which variables belong to the set $\textsc{rel}\,(e)$ of the variables *relevant* in the evaluation of e. As already pointed out, standard dependency calculi compute $\textsc{rel}\,(e)$ as $\textsc{vars}\,(e)$. In the following definition, e is said to *semantically* depend on x (written $x \rightsquigarrow_{\mathrm{S}} e$) if the evaluation of e may change depending on x, i.e., if there exist two values for x leading to different values of e.

DEFINITION 4.1 (Semantic dependencies, *Dep*). *Let* $x \in$ VAR, $Y \subseteq$ VAR.

$$\begin{aligned} x \rightsquigarrow_{\mathrm{S}} e \quad &\Leftrightarrow \quad \exists \sigma_1 . \sigma_2 \in \Sigma .\ \forall y \neq x .\ \sigma_1(y) = \sigma_2(y) \\ &\qquad \wedge\ \mathcal{E}\,[\![e]\!]\,(\sigma_1) \neq \mathcal{E}\,[\![e]\!]\,(\sigma_2) \\ Y \rightsquigarrow_{\mathrm{S}} e \quad &\Leftrightarrow \quad \exists y \in Y .\ y \rightsquigarrow_{\mathrm{S}} e \end{aligned}$$

The formulation of $x \rightsquigarrow_{\mathrm{S}} e$ *can be rewritten as*

$$\exists \sigma \in \Sigma, v_1, v_2 \in \mathbb{V}.\ \mathcal{E}\,[\![e]\!]\,(\sigma\,[x \leftarrow v_1]) \neq \mathcal{E}\,[\![e]\!]\,(\sigma\,[x \leftarrow v_2])$$

By using this notion of dependency, we can characterize the subset of VARS (e) containing exactly only, and all, the variables which are *semantically* relevant for the evaluation of e. This way, we obtain a notion of dependency which implies deriving more precise slices, namely, we are able to remove statements a standard, syntactic analysis would leave. Consider the following running example.

EXAMPLE 4.2. *Consider the program:*

$$\mathcal{P} \equiv s_x\ ;\ s_y\ ;\ s_w\ ;\ z := w + y + 2x^2 - w\ ;$$

where s_x, s_y and s_w define, resp., x, y and w, and $e \overset{def}{=} w + y + 2x^2 - w$. We want to compute the slice $\mathcal{P}'$ of $\mathcal{P}$ affecting the final value of z (i.e., the slicing criterion *S is the final value of z). If we consider the standard notion of slicing, then it is clear that we can erase s_w without changing the final result for z. Considering the standard PDG approach, we would have a dependency between z and w, since w is used where z is defined. Consequently, the slice obtained by applying this form of dependency would leave the program unchanged. On the other hand, if the semantic dependency is considered, then the evaluation of e does not depend on the possible variations of w, which implies that we are able to erase s_w from the slice.*

Consider now abstract properties of data: in this context, the picture is quite different. The abstract calculus of dependencies is a weaker version of its standard (concrete) counterpart: it is often the case that e (semantically) depends, with respect to the considered abstract property, on a strict subset of VARS (e) (and, also, of REL (e), defined w.r.t. $x \rightsquigarrow_{\mathrm{S}} e$). Given an abstract property ρ, e depends on x w.r.t. the property ρ (written $x \overset{\rho}{\rightsquigarrow} e$) if, although all the variables, but x, have the same property, the evaluations of e may have a different property. In other words, the property ρ of e does not depend on x when the rest of the input state is fixed w.r.t. ρ. Note that this definition is ambiguous since it does not specify how the expression is evaluated in terms of the abstract input. We suppose the analysis of abstract dependencies to be based on the *concrete* semantics, as it happens, for example, in program slicing, where abstract dependency properties are derived by analyzing the control flow graph of the program, i.e., a representation of the program relying on its concrete semantics. In this case, we define the so called *narrow* abstract dependencies.

(Narrow) abstract dependencies

The following definition is called *narrow* since it follows the same philosophy as narrow abstract non-interference [9], where we consider abstractions for observing input and output, but these abstractions are observations of the *concrete* evaluation of the expression[4].

DEFINITION 4.3 (*Ndep*). *Let* $\rho \in$ UCO$(\wp(\mathbb{V}))$.

$$\begin{aligned} x \overset{\rho}{\rightsquigarrow}_{\mathrm{N}} e \Leftrightarrow\ & \exists \sigma_1, \sigma_2 \in \Sigma . \forall y \neq x .\ \rho(\sigma_1(y)) = \rho(\sigma_2(y)) \\ & \wedge\ \rho\left(\mathcal{E}\,[\![e]\!]\,(\sigma_1)\right) \neq \rho\left(\mathcal{E}\,[\![e]\!]\,(\sigma_2)\right) \end{aligned}$$

Note that ρ is always applied to singletons ($\rho(v)$ stands for $\rho(\{v\})$). Therefore, the focus is merely on how domains behave on single values. As an important result, we have $x \overset{\rho'}{\rightsquigarrow}_{\mathrm{N}} e \Leftrightarrow x \overset{\rho''}{\rightsquigarrow}_{\mathrm{N}} e$ for every ρ' and ρ'' inducing the same partition on singletons. Since it is straightforward to note that $x \overset{\rho}{\rightsquigarrow}_{\mathrm{N}} e$ is affected only by $\Pi(\rho)$, rather than by ρ itself, in the following we only consider, without loss of generality, partitioning domains.

Note that *Ndep* is reasonable from a practical point of view, since it fits in the framework of several analysis techniques, and in particular in program slicing. However, this notion suffers the problem of adding some *false* dependencies; this is due to the fact that also the concrete value of variables different from x can vary in the input changing the value of e, although their property ρ is fixed.

EXAMPLE 4.4. *Consider the program in Example 4.2 and consider the* PAR *property. If we compute the set of relevant variables for the parity of e, then we can note that, still, we keep the independency from w, but also that the parity of e does not even depend on any possible variation of x. Therefore, the outlined non-standard version working on properties may lead to a smaller (more precise) slice. In fact, if we consider the abstract dependencies with $\rho =$* PAR*, the independency $[z \ltimes x]$ holds after the assignment, since the parity of the evaluation of e does not change by modifying x, which means that the slice can erase the statement s_x.*

At a first sight, in the previous example, the independency of the parity of e from x appears to be due simply to the fact that, in e, the parity of x is a constant, since $2x^2$ is always even. However, a deeper analysis would note that we can look simply at the abstract value of x only because the operation involved (the sum) in the evaluation is complete [7, 11] w.r.t. the abstract domain considered (PAR). The following example, shows that, if we deal with operations which are not complete for the considered abstract domain, then it is not sufficient to look at the abstract value of the variable for deriving the dependencies.

EXAMPLE 4.5. *Consider the program in Example 4.2, and the* SIGN *domain. In this case, even if the sign of the expression $2x^2$ is constantly positive, still the sign of z might change due to a concrete variation in x (e.g., consider $y = -4$ and two executions in which x is resp. 1 and 5). Therefore, x has to be considered as relevant to S although the sign of $2x^2$ (the only sub-expression containing x) is constant. This can be derived also by considering the logic of independencies, since, by varying the value of x, we indeed can change the sign of e. These issues will play a rôle in defining a different notion of abstract dependency (Sec. 7).*

The key point in the examples above is that, by considering abstract dependencies, we are at the same time abstracting the criterion, i.e., we aim to detect which variables affect a property of e (e.g., sign) instead of the exact value. The problem is that the correspondence between abstraction of the dependency and abstraction of the criterion is not so straightforward.

[4] Note that, in the following of the paper, in order to simplify the notation and the construction, we consider the same property in input and in output for the evaluation of e. Nevertheless, the whole construction can be generalized by using two different abstractions.

The problem is that, even if we consider semantic [abstract] dependencies, we still could not be able to reach the most precise slice for a given criterion. This loss of precision occurs whenever we have to deal with control statements which implicitly remove some semantic dependencies in a way which is not detectable by looking only at semantic [abstract] dependencies for expressions. Consider, for example

$$\textbf{if } (y + 2x \textit{ mod } 2) == 0 \textbf{ then } w := 0 \textbf{ else } w := 0$$

then we can erase the dependency between the guard and x, since the value of the guard does not depend on x, but we cannot remove the dependency between w and y, annulled by the semantics of the **if** (the variation of y cannot change the final value of w). The only way to detect this independency is to realize that the final value of w is invariant w.r.t. the evaluation of the guard, and this requires more specific semantic analyses of statements, which go beyond the aim of this paper.

Hence, what we propose here is a *sound* approximation of abstract slicing, i.e., slicing w.r.t. abstract criteria, obtained by using abstract dependencies for *pruning* the syntactic PDG, since many dependencies only hold at the syntactic level. As underlined in the example above, the problem is that in this framework we are only able to prune flow dependence edges, while the control dependence ones need more specific analyses for being erased.

Now, computing a slice consists in deleting statements which are not reachable in the PDG, starting from the information provided by the slicing criterion. It is easy to see that abstract dependencies lead to smaller slice (therefore, they imply a *weaker* notion of slicing), because a statement is less likely to be reachable (i.e., to be kept in the slice) in the pruned PDG.

5. A constructive approach to abstract dependencies

This section discusses a constructive way to compute abstract dependency. By means of the (domain-dependent) definition of operations on abstract values, it is possible to automatically obtain (an over-approximation of) the set of relevant variables.

An algorithm describing this approach is provided. Importantly, it must be pointed out that, as far as simple numerical domains are concerned, the interested reader will not find dramatic computational improvements w.r.t. the *brute force*, exponential approach which takes all possible abstract values for each variable, and actually *goes into* the quantifiers explicitly or implicitly involved in Def. 4.3. In fact, computational effort can be saved if the domain is big and many variables are involved, and a significant part of them is found to be irrelevant. Yet, this work mainly focuses on giving an insight on how abstract dependencies work, and provides a reasonable basis for applications where domains can be really huge and complicated.

5.1 Checking *Ndep*

We discuss the main ideas for constructively computing narrow dependencies. The algorithm in Fig. 2 tries to find, incrementally and starting from $\emptyset$, a set of variables which are provably not relevant to the studied expression. This is done by checking whether a change in such variables does not make a difference in the evaluation. Finally, the set-complement of the maximal set of variables for which irrelevance can be proved is returned.

First, note that, in the realm of static analysis, the concrete semantics cannot be used directly as it appears in Def 4.3. Hence, we will define narrow dependencies in terms of the abstract semantics $\mathcal{E}\,[\![\cdot]\!]^\rho$:

DEFINITION 5.1 (*Atom-Adep*). $x \overset{\rho}{\rightsquigarrow}_{\text{AT}} e \Leftrightarrow$

$$\exists \sigma_1, \sigma_2 \in \Sigma.\ \forall y \neq x.\ \rho(\sigma_1(y)) = \rho(\sigma_2(y)) \wedge \\ \neg\text{ATOM}\left(\mathcal{E}\,[\![e]\!]^\rho\,(\rho(\sigma_1) \cup \rho(\sigma_2))\right)$$

Being the domains partitioning, the non-atomicity requirement on the evaluations of e amounts to say that all concrete evaluations (which yield singletons) on $\rho(\sigma_1) \cup \rho(\sigma_2)$ would not be abstracted by the same abstract value (this is the crucial issue in *Ndep*), i.e., σ_1 and σ_2 may lead to different values for e. Indeed, Definitions 4.3 and 5.1 are equivalent:

THEOREM 5.2. *For every e and x, $x \overset{\rho}{\rightsquigarrow}_{\text{N}} e$ iff $x \overset{\rho}{\rightsquigarrow}_{\text{AT}} e$.*

Proof. Suppose $x \overset{\rho}{\rightsquigarrow}_{\text{N}} e$ does not hold; we prove that also $x \overset{\rho}{\rightsquigarrow}_{\text{AT}} e$ is false. The hypothesis amounts to say that, if there exist σ_1, σ_2 such that $\forall y \neq x.\ \rho(\sigma_1(y)) = \rho(\sigma_2(y))$, then $\rho(\mathcal{E}\,[\![e]\!]\,(\sigma_1)) = \rho(\mathcal{E}\,[\![e]\!]\,(\sigma_2))$. Note that

$$\begin{aligned} \rho\left(\mathcal{E}\,[\![e]\!]\,(\rho(\sigma_1(y)))\right) &= \rho\left(\textstyle\bigcup_{y' \in \rho(\sigma_1(y))} \mathcal{E}\,[\![e]\!]\,(y')\right) \\ &= \rho\left(\mathcal{E}\,[\![e]\!]\,(\sigma_1(y))\right) \end{aligned}$$

by the additivity of the abstract domain. Since $\sigma_1(y)$ is a singleton and ρ is partitioning, we have that $\rho(\mathcal{E}\,[\![e]\!]\,(\sigma_1(y)))$ is an atom of ρ. Analogously, we obtain that $\rho(\mathcal{E}\,[\![e]\!]\,(\sigma_2(y)))$ is an atom of ρ. Moreover, these two sets are the same by the hypothesis that *Ndep* does not hold. Hence, their union is an atom and is equal to $\mathcal{E}\,[\![e]\!]^\rho\,(\rho(\sigma_1) \cup \rho(\sigma_2))$ by additivity of all the functions involved.

On the other side, suppose $x \overset{\rho}{\rightsquigarrow}_{\text{AT}} e$ does not hold: we prove that neither *Ndep* holds. Suppose that there exist σ_1, σ_2 such that $\forall y \neq x.\ \rho(\sigma_1(y)) = \rho(\sigma_2(y))$; the hypothesis guarantees the atomicity of $\mathcal{E}\,[\![e]\!]^\rho\,(\rho(\sigma_1) \cup \rho(\sigma_2))$. By additivity, this fact implies that both $\rho(\mathcal{E}\,[\![e]\!]\,(\sigma_1(y)))$ and $\rho(\mathcal{E}\,[\![e]\!]\,(\sigma_2(y)))$ are atomic and equal, otherwise their union could not be an atom. Therefore, we have the thesis, i.e., *Ndep* does not hold. $\square$

In general, the interest is in finding the set of relevant variables, rather than verifying if a given x is relevant. To deal with abstract computations, we provide operations on ρ (e.g., in PARSIGN, rules like $[\textbf{odd}] + [\textbf{odd}] = [\textbf{even}]$ or $[\textbf{poseven}] * [\top] = [\textbf{even}]$, and an abstract $\sqcup$ to model $\cup$ on states). This leads to a computable $\mathcal{E}\,[\![e]\!]^\rho$, which, however, may lose some information, and also the *Ndep*/*Atom-Adep* equivalence. One direction of the implication of Theorem 5.2 still holds: if a dependency $x \overset{\rho}{\rightsquigarrow}_{\text{N}} e$ exists, then $x \overset{\rho}{\rightsquigarrow}_{\text{AT}} e$ is detected as true. This is a *soundness* condition, if the purpose is to over-approximate the set of relevant variables as required, e.g., in computing correct slices.

Dependencies are computed according to *Atom-Adep*, in order to approximate *Ndep*. In the brute force approach, *Atom-Adep* is verified by checking $\text{ATOM}\left(\mathcal{E}\,[\![e]\!]^\rho\,(\varepsilon)\right)$ for every ε atomic (i.e., abstraction of a singleton $\{\sigma\}$).

EXAMPLE 5.3. *Let ρ = PARSIGN. In order to compute the set of ρ-dependencies on $e\,[x, y, z]$, we must compute $\mathcal{E}\,[\![e]\!]^\rho$ on ev-*

ery possible atomic value[5] *of x, y and z, i.e., $\mathcal{E}\llbracket e\rrbracket^\rho$ must be computed 4^3 times. y is* not *relevant to e if, for any $V_x, V_z \in$ $\textsc{Atoms}(\rho)$, there exists U atomic s.t. $\forall V \in \textsc{Atoms}(\rho).\ U = \mathcal{E}\llbracket e\rrbracket^\rho(\langle V_x, V, V_z\rangle)$. This amounts to say that changing y does not affect e.*

Indeed, it is possible to be smarter, by taking some arrangements into account:

- *Excluding states:* consider dependencies of $e[x,y,z]$ in Ex. 5.3, computed at the program point p. Suppose $\mathcal{P}\llbracket s\rrbracket_p$ infers $\varepsilon_p(y) = [\mathbf{posodd}]$ (Sec. 2). Then, we only need to consider states of the form $\langle V_x, [\mathbf{posodd}], V_z\rangle$ as inputs for $\mathcal{E}\llbracket e\rrbracket^\rho$ at p.
- *Computing on non-atomic states:* let $E = \{[\mathbf{poseven}], [\mathbf{negeven}]\}$ and $O = \{[\mathbf{posodd}], [\mathbf{negodd}]\}$. In this case,
$$\forall V' \in E, V'' \in O.\ \mathcal{E}\llbracket e\rrbracket^\rho(\langle V_x, V', V''\rangle) \le U$$
is implied by the more general result
$$\mathcal{E}\llbracket e\rrbracket^\rho(\langle V_x, [\mathbf{even}], [\mathbf{odd}]\rangle) \le U$$
since E and O are partitions, respectively, of $[\mathbf{even}]$ and $[\mathbf{odd}]$, and $\mathcal{E}\llbracket e\rrbracket^\rho$ is monotone: $\varepsilon' \le \varepsilon''$ implies $\Rightarrow \mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon') \le \mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon'')$. This means that results obtained on ε can be used on $\varepsilon' \le \varepsilon$.

Let the set $[\varepsilon|X]$ denote all the states $\varepsilon' \le \varepsilon$ s.t. $\forall x \in X.\ \varepsilon'(x) = \varepsilon(x)$, and $\forall y \notin X.\ \textsc{Atom}(\varepsilon'(y))$. To prove e not dependent on x, we need to prove $\textsc{Atom}\left(\mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon_x)\right)$, for any $\varepsilon_x \in [\varepsilon_p|\{x\}]$. This amounts to say that any variation in x does not lead to an observable variation in e, whenever all the other variables are fully specified as atoms. Given e and an atom U, the *atomicity condition* $\mathcal{A}^U_e(\varepsilon)$ holds iff $\mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon)$ gives U, or there exists a covering $\{\varepsilon_1, .., \varepsilon_k\}$ of ε such that $\mathcal{A}^U_e(\varepsilon_i)$ holds for every i. Importantly, $\mathcal{A}^U_e(\varepsilon)$ implies that $\rho(\{\mathcal{E}\llbracket e\rrbracket(\sigma)|\sigma \in \varepsilon\})$ is an atom, and the second disjunct helps if, due to a possible loss of information, $\mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon) > U$ although $\forall\varepsilon' < \varepsilon.\ \mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon') = U$.

EXAMPLE 5.4. *Let $e \equiv x{*}x{+}1$ and $\rho = \textsc{Sign}$. $\mathcal{E}\llbracket e\rrbracket^\rho$ follows the usual rules on $*$ and $+$: $[\mathbf{pos}] * [\mathbf{pos}] = [\mathbf{pos}]$, $[\mathbf{neg}] * [\mathbf{neg}] = [\mathbf{pos}]$, $[\top]{*}[\top] = [\top]$, $[\mathbf{pos}]{+}[\mathbf{pos}] = [\mathbf{pos}]$, $[\top]{+}[\mathbf{pos}] = [\top]$. This is enough to compute $\mathcal{E}\llbracket e\rrbracket^\rho(\langle[\mathbf{neg}]\rangle) = \mathcal{E}\llbracket e\rrbracket^\rho(\langle[\mathbf{pos}]\rangle) = [\mathbf{pos}]$. Yet, although $[\top] = [\mathbf{pos}] \cup [\mathbf{neg}]$, the general result $\mathcal{E}\llbracket e\rrbracket^\rho(\langle[\top]\rangle) = [\mathbf{pos}]$ cannot be proven with these rules, since $[\top] * [\top] + [\mathbf{pos}] = [\top]$. In this case, the result $\mathcal{A}^{[\mathbf{pos}]}_e(\langle[\top]\rangle)$ can be proven because of the second condition on the covering $\{\langle[\mathbf{pos}]\rangle, \langle[\mathbf{neg}]\rangle\}$.*

By *Atom-Adep*, to prove the non-relevance of x it is enough to have $\exists V.\mathcal{A}^V_e(\varepsilon_x)$ for every $\varepsilon_x \in [\varepsilon_p|\{x\}]$, where it is not required to have the same atom for all states. We define X-coverings of states: a covering $\{\varepsilon_1..\varepsilon_k\}$ of ε is an *X-covering* if (i) for every $x \in X$, $\varepsilon_i(x) = \varepsilon(x)$ holds for every i; (ii) for every $y \notin X$, $\varepsilon_i(y)$ is $\varepsilon(y)$ or one of its direct sub-values, i.e., a $V < \varepsilon(y)$ s.t. $\nexists V'.\ V < V' < \varepsilon(y)$. The assertion $\mathcal{A}'_e(\varepsilon, X)$, where $X \subseteq \textsc{Var}$, holds iff $\exists U.\ \mathcal{A}^U_e(\varepsilon)$, or there exists an X-covering $\{\varepsilon_1..\varepsilon_k\}$ of ε, such that $\forall i.\ \mathcal{A}'_e(\varepsilon_i, X)$. Intuitively, an X-covering is a set of restrictions on a state, which do not involve X. Clearly, the condition for the non-relevance of a set X is related to the definition of X-covering, since $[\varepsilon_p|X]$ can be obtained by repeatedly applying to ε_p (and the newly obtained states) the X-covering operation. Indeed, $\mathcal{A}'_e(\varepsilon, X)$ implies $\textsc{Atom}\left(\mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon_X)\right)$ on every $\varepsilon_X \in [\varepsilon|X]$ and, therefore, the non-relevancy of X.

[5] Atoms in ρ are $[\mathbf{poseven}]$, $[\mathbf{posodd}]$, $[\mathbf{negeven}]$, and $[\mathbf{negodd}]$; since this is a partition of concrete values, we describe all concrete inputs by computing $\mathcal{E}\llbracket e\rrbracket^\rho$ on atoms.

```
function FINDNDEPS { // e is left implicit
  nonDep := $\emptyset$; // can be modified by prove()
  PROVE($\varepsilon_p$, VARS($e$));
  return VARS($e$) $\setminus$ nonDep; } // relevant vars

procedure PROVE($\varepsilon$, $X$) {
  if ($\mathcal{A}'_e(\varepsilon, X)$) then {
    nonDep := nonDep $\cup$ $X$;
  } else foreach ($x \in X$) {
    PROVE($\varepsilon$, $X \setminus \{x\}$); } }
```

Figure 2. The FINDNDEPS algorithm

The algorithm FINDNDEPS (Fig. 2) starts by trying to prove $\mathcal{A}'_e(\varepsilon_p, \textsc{Var})$; since $\{\varepsilon\}$ is the only $\textsc{Var}$-covering of any ε, this condition is only satisfiable if $\mathcal{A}^U_e(\varepsilon_p)$ holds, i.e., if e depends on no variables. Otherwise, the set X is decreased non-deterministically until some $\mathcal{A}'_e(\varepsilon_p, X)$ is proven. $\mathcal{A}'_e(\varepsilon_p, X)$ means that an atomic value for e was obtained without the need of restricting X variables; therefore, X only contains non-relevant variables.

PROPOSITION 5.5. *If $\mathcal{A}'_e(\varepsilon_p, X)$ can be proven, then there is no $x \in X$ such that $x \overset{\rho}{\rightsquigarrow}_{\mathrm{AT}} e$.*

Proof. Consider the definition of $\mathcal{A}'_e$: if $\mathcal{A}^U_e(\varepsilon_p)$ holds for some atom U, then there are no dependencies. Otherwise, let ε_i be one of the states belonging to the X-covering; for every $\sigma_i \in \varepsilon_i$, $\mathcal{E}\llbracket e\rrbracket(\sigma_i)$ belongs to the same atom, i.e., there is no way to distinguish between two computations. This means that it is possible to change X variables to any value, without changing the property of e. Since this is true for every ε_i, and these states are a covering of ε_p, the thesis follows. $\square$

Importantly, the assertions $\mathcal{A}'_e(\varepsilon_p, X)$ and $\mathcal{A}'_e(\varepsilon_p, Y)$ guarantee $X \cup Y \overset{\rho}{\rightsquigarrow}_{\mathrm{AT}} e$ not to hold, even if $\mathcal{A}'_e(\varepsilon_p, X \cup Y)$ cannot be directly proven. The final result of FINDNDEPS is $\textsc{vars}(e) \setminus Z$, where Z is the union of all sets Z_i such that $\mathcal{A}'_e(\varepsilon_p, Z_i)$ can be proved. It is an over-approximation of relevant variables $\textsc{rel}(e)$.

FINDNDEPS may deal, in principle, with *infinite* domains, since non-dependency results can be possibly proven without exploring the entire state-space; in fact, if $\mathcal{A}^U_e(\varepsilon)$ can be proven, then it is not needed to descend into the (possibly infinite) set of sub-states of ε. This is not possible in the brute force approach. It is also straightforward to add *bounds* in order to stop refining states if some computational threshold has been reached.

6. Dependencies erasure in the abstract framework

The problem of computing abstract dependencies can be observed from another point of view: given e and a set X of variables, we

```
$\rho$ := $\rho_0$; // the initial domain
repeat {
  inputQueue := [$\varepsilon_p$]; // one-element queue
  while (notEmpty(inputQueue)) {
    $\varepsilon$ := extract(inputQueue);
    if ($\nexists V.\ \mathcal{A}_e^V(\varepsilon)$) then {
      if (ATOM($\varepsilon$)) then { // on VARS$(e) \smallsetminus X$
        // at this point, $V$ is not atomic
        $V$ := $\mathcal{E}[\![e]\!]^\rho(\varepsilon)$;
        $\rho$ := ATOMIZE$(\rho, V)$;
        // the queue still has 1 element
        inputQueue := [$\varepsilon_p$];
      } else { // $\{\varepsilon_1..\varepsilon_k\}$ is an $X$-covering of $\varepsilon$
        foreach($i$) {
          insertInQueue(inputQueue, $\varepsilon_i$)}}}}
} until ($\rho$ has not been modified in the while loop);
return $\rho$; // the domain s.t. $X \overset{\rho}{\rightsquigarrow}_{\mathrm{N}} e$ does not hold
```

Figure 3. The EDEP algorithm.

may be interested in soundly approximating the most concrete ρ such that $X \overset{\rho}{\rightsquigarrow}_{\mathrm{N}} e$ does not hold. This can be accomplished by repeatedly simplifying an initial domain ρ_0 in order to eliminate abstract values which are *responsible* for dependencies. In order to avoid dependencies on X, we should have $\mathcal{A}'_e(\varepsilon_p, X)$, i.e., $\mathcal{A}_e^V(\varepsilon_X)$ should hold for any $\varepsilon_X \in [\varepsilon_p|X]$. If this does not hold for some ε, then ρ is modified to obtain the atomicity of $V = \mathcal{E}[\![e]\!]^\rho(\varepsilon)$.

We design a simple algorithm EDEP(e, ρ_0, X) (Fig. 3), which repeatedly checks if there exists V s.t. $\mathcal{A}_e^V(\varepsilon)$. Initially, the current state ε is ε_p; then, it is progressively specialized to states belonging to one of its X-coverings, until one of the following holds:

- $\mathcal{A}_e^V(\varepsilon)$; in this case, ρ is precise enough to exclude dependencies on X in ε, and is not modified;
- ε cannot be refined anymore (it is atomic on VARS$(e) \smallsetminus X$) but V is non-atomic; in this case, ρ needs to be simplified in order to obtain ATOM(V).

States are processed by means of a queue; we stop when all the states have been consumed without any modification to ρ, i.e., when no non-atomic V has been found. As in FINDNDEPS, states are progressively restricted, and computations on $\mathcal{E}[\![e]\!]^\rho(\varepsilon)$ are avoided if the desired property already holds for $\varepsilon' > \varepsilon$.

The simplifying operator on ρ and V is a *domain transformer*, and works by removing abstract values in order to obtain ATOM(V). Formally,

$$\rho' = \text{ATOMIZE}(\rho, V) \overset{\text{def}}{=} \{U \in \rho \mid V \cap U = \bot \ \vee\ V \le U\}$$

The final ρ is an approximation of the most precise ρ' s.t. $X \overset{\rho'}{\rightsquigarrow}_{\mathrm{N}} e$ is false:

THEOREM 6.1. *$\rho \overset{def}{=}$ EDEP(e, ρ_0, X) makes e not narrow dependent on X. In other words: the final ρ satisfies non-dependency of e on X, that is, for every ε which is atomic on* VARS$(e) \smallsetminus X$, *$\mathcal{E}[\![e]\!]^\rho(\varepsilon)$ is atomic.*

Proof. The algorithm halts if, in processing ε_p, ρ is not changed. Processing ε_p involves computing $\mathcal{E}[\![e]\!]^\rho$ on sub-states when required, in order to prove the atomicity property on every concrete state represented by ε_p (we exploit monotonicity of $\mathcal{E}[\![e]\!]^\rho$ on states). This is precisely obtained if every state is removed from the queue before any modification to ρ occurs. □

On the practical side, the loss of precision in abstract computations may lead to remove more abstract values than strictly necessary from the semantic point of view.

It is important to note that EDEP works as long as $\mathcal{A}_e$ can be computed on the initial domain (in this case, no problems arise in subsequent computations, since the complexity of ρ can only decrease). This can possibly happen even if ρ_0 is infinite (see the last part of the previous section). Moreover, differently from FINDNDEPS, there is no reasonable trivial counterpart, since brute force would be really impractical.

7. An application to secure information flow

As suggested before, our approach to abstract dependencies (and abstract slicing) has been inspired by a recent abstract interpretation-based model of non-interference in language-based security, namely, abstract non-interference [9]. Non-interference [12, 19] is an information flow property of programs, enforcing confidentiality of data. Let program data be divided into a public part `L` and a private part `H`; non-interference is satisfied if an external user is not able to acquire information about the initial value of variables classified as `H` by only observing the final value of variables classified as `L`. It is well known that confidentiality can be modeled by means of dependency relations among the different components of a program [6, 3, 2]. This model makes clear the strong relation existing between non-interference and dependencies. In a recently proposed model, *abstract non-interference* [9], abstractions come into play in modeling the observational power of attackers. We show, in this section, that the relation existing between slicing/dependencies and non-interference is even stronger in this abstract context. In particular, computing abstract dependencies, provides a technique for computing abstract non-interference certifications. This means that our approach to abstract slicing, is not simply a pure generalization of a well-known technique, but provides new insights in links existing between different computer science fields.

7.1 Abstract dependencies.

In introducing narrow dependencies (Sec. 4), we mention the possibility of choosing the semantics which is used to evaluate expressions. Taking an abstract semantics leads to an alternative version of abstract dependencies, which is the one that we have to consider for reasoning on the most general notion of abstract non-interference.

To this end, we formalize the same notion for abstract dependencies in a more general way, by considering the abstract evaluation of e in ρ (i.e., the best correct approximation of $\mathcal{E}[\![]\!]^\rho$):

DEFINITION 7.1 (*Adep*). *Let $\rho \in$ UCO$(\wp(\mathbb{V}))$.*

$$x \overset{\rho}{\rightsquigarrow}_{\mathrm{A}} e \iff \exists \sigma_1, \sigma_2 \in \Sigma.\ \forall y \neq x.\ \rho(\sigma_1(y)) = \rho(\sigma_2(y)) \\ \wedge\ \mathcal{E}[\![e]\!]^\rho(\rho(\sigma_1)) \neq \mathcal{E}[\![e]\!]^\rho(\rho(\sigma_2))$$

Note that, false dependencies are be avoided in this approach (indeed, this has been one of the main reasons why the related, non-narrow notion of abstract non-interference was introduced). Clearly, *Adep* makes the assumption that the analyzer can compute directly the abstract semantics (in the following, a computable version of $\mathcal{E}\llbracket\rrbracket^\rho$ will be used). It is worth noting that *Adep* is a stronger notion than *Ndep* [9], in the sense that *abstract* dependencies are less likely to occur than their *narrow* counterpart: $x \overset{\rho}{\rightsquigarrow}_{\text{A}} e$ implies $x \overset{\rho}{\rightsquigarrow}_{\text{N}} e$. Moreover, in both *Ndep* and *Adep*, the more precise the chosen domain, the more dependencies may occur.

7.2 Dealing with *Adep*

The FINDNDEPS algorithm can be modified in order to account for *Adep*. This is, in a sense, a stronger property than *Ndep* (Sec. 4). The main difference between *Ndep* and *Adep* is the choice of the semantics used to observe the program execution. In the latter, we suppose to analyze dependencies in the abstract semantics $\mathcal{E}\llbracket e\rrbracket^\rho$. To compute *Adep*, we do not need to define a new implementable version of abstract dependencies, as we did with *Ndep*. Instead, we can change FINDNDEPS to obtain a second version FINDADEPS, as shown in the next two paragraphs.

7.2.1 Comparing executions.

The atomicity of results (Def. 5.1) is no longer required: it is enough to guarantee that $\mathcal{E}\llbracket e\rrbracket^\rho$ is the same in every state ε' *compatible* with ε (i.e., $\varepsilon' \le \varepsilon$); This leads to a weaker definition of $\mathcal{A}_e^U(\varepsilon)$:

$$\begin{array}{rcl}
\mathcal{A}_e^U(\varepsilon) & \equiv & \mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon) \doteq U \\
& \vee & \exists \text{ covering } \{\varepsilon_1,..,\varepsilon_k\}.\ \forall i. \mathcal{A}_e^U(\varepsilon_i) \\
\mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon) \doteq U & \equiv & \mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon) = U \\
& \wedge & \nexists \varepsilon' < \varepsilon.\ \mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon') < U
\end{array}$$

The predicate $\doteq$ can be defined compositionally on the structure of e, and, obviously, relies on our knowledge of the abstract domain (i.e., it is necessary to know how abstract operations behave).

7.2.2 Replacing expressions.

During computations on a state ε, an expression e may be replaced by any e' which is *equivalent* w.r.t. $\mathcal{E}\llbracket e\rrbracket^\rho$ on every $\varepsilon' \le \varepsilon$. In this case, the replacement is useful if the new expression is simpler. For example, $e_1 + e_2$ can be replaced by $e_1 + U$ in ε, provided $\mathcal{E}\llbracket e_2\rrbracket^\rho(\varepsilon) \doteq U$. To replace an expression by a constant value may simplify the original expression and, possibly, get rid of some variables. The substitution is parametric on the state, hence its validity cannot be extended to larger states: this is important when states are restricted (e.g., in computing $\mathcal{A}_e^U(\varepsilon)$ and $\mathcal{A}'_e(\varepsilon, X)$).

EXAMPLE 7.2. *Let $\varepsilon'' < \varepsilon'$. In order to be equivalent on ε'', two expressions e_1 and e_2 should be such that $\forall\varepsilon \le \varepsilon''.\ \mathcal{E}\llbracket e_1\rrbracket^\rho(\varepsilon) = \mathcal{E}\llbracket e_2\rrbracket^\rho(\varepsilon)$. Indeed, this condition does not imply the more general one on ε', since there may exist $\varepsilon \le \varepsilon'$ s. t. $\mathcal{E}\llbracket e_1\rrbracket^\rho(\varepsilon) \neq \mathcal{E}\llbracket e_2\rrbracket^\rho(\varepsilon)$.*

The algorithm can exploit this additional information by collecting pairs (e_0, ε) whenever computations show e_0 to be equivalent to e on ε (e.g., when $e_0 = V$ and $\mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon) \doteq V$ is verified). In further computations, if the analysis is performed on a state $\varepsilon' \le \varepsilon$ (for example, in proving $\mathcal{A}_e^U(\varepsilon')$), then e_0 can be safely used instead of e, thus resulting, generally, in an efficiency improvement.

It is worth noting that the replacement of expressions cannot be done when we consider *Ndep*, because the concrete semantics could be able to distinguish two executions even if $\mathcal{E}\llbracket\cdot\rrbracket^\rho$ cannot.

EXAMPLE 7.3. *Let $e \equiv y + 2x^2$ be analyzed in* PARSIGN *w.r.t. narrow dependency. Since $2x^2 \doteq [\mathbf{poseven}]$, we may think it safe to replace e by $y + [\mathbf{poseven}]$. However, given a fixed value for y, there exist two values of x such that the final value can be distinguished. Indeed, this is correctly accounted for in* FINDNDEPS*: evaluating $\mathcal{E}\llbracket e\rrbracket^\rho(\varepsilon)$ does not yield an atom, so that* FINDNDEPS *does not exclude dependencies on x.*

7.3 Enforcing abstract non-interference.

In abstract non-interference, malicious external users can only see input and output data up to a degree of precision which is described by two abstract domains η and ρ. ANI models the property that an attacker observing η of public input and ρ of public output is unable to break the secrets of a program $\mathcal{P}$ (we write $(\eta)\mathcal{P}(\rho)$ if $\mathcal{P}$ satisfies this security requirement).

Giacobazzi and Mastroeni [10] give a systematic method for enforcing ANI, i.e., a logic whose assertions take the form $(\eta)\ s\ (\rho)$ for a statement s and two domains η and ρ. Consider, in particular, two rules (the second is specialized on $\eta = \rho$), which allow to derive ANI properties for expressions and assignments[6].

$$\mathbf{A1} : \frac{(\eta)\ \llbracket e\rrbracket\ (\text{ID}) \sqsubseteq \rho}{(\eta)\ e\ (\rho)} \qquad\qquad \mathbf{A4} : \frac{(\rho)\ e\ (\rho)\,,\quad x \text{ public}}{(\rho)\ x := e\ (\rho)}$$

The expression $(\eta)\ \llbracket e\rrbracket\ (\text{ID})$ in **A1** denotes the *secret kernel* [9] (on expressions) of e w.r.t. η, that is, the most precise domain ρ_k which cannot break secrets on e when the domain on input is η (i.e., such that $(\eta)e(\rho_k)$). Importantly, an approximation of the secret kernel can be obtained by the EDEP algorithm (Sec. 6), which provides a mechanizable version of the derivation, close to the original, semantic one [9]. Rule **A4** means that an assignment to a public variable is secure if the assigned expression does not depend on any private variables w.r.t. ρ (i.e., $(\rho)\ e\ (\rho) \Leftrightarrow \neg((\text{VARS}^{\text{H}}\ (e)) \overset{\rho}{\rightsquigarrow} e)$[7]).

Whenever we are interested in having the same observation both in input and in output, we can note that, if $(\rho)\ \llbracket e\rrbracket\ (\text{ID}) \sqsubseteq \rho$, then $(\rho)\ e\ (\rho)$. Indeed, $(\rho)\ \llbracket e\rrbracket\ (\text{ID})$ returns the most concrete output observation making the expression evaluation independent from the private variables when the input observation is ρ; if this is computed as a fixpoint, then $\text{LFP}_{\text{ID}}\ (\lambda X.\ (X)\ \llbracket e\rrbracket\ (\text{ID}))$ characterizes a harmless attacker observing the same property in input and in output, i.e., $(\rho)\ e\ (\rho)$. Unfortunately, the function is not monotone, therefore we are not sure to find exactly the *most powerful* attacker [9].

The fixpoint computation is incorporated in the EDEP algorithm (possibly, not starting from the ID domain). Therefore, it is possible to use the abstract dependency computation in order to approximate the ANI property for assignments and, consequently, make the whole derivation systematic. In particular, we may replace **A1**[8] by

$$\mathbf{A1}' : \frac{\rho_e \sqsubseteq \rho}{(\rho_e)\ e\ (\rho)} \qquad \text{where} \qquad \rho_e \overset{\text{def}}{=} \text{EDEP}\left(e, \text{ID}, \text{VARS}^{\text{H}}\ (e)\right)$$

The resulting domain ρ_e also characterizes an attacker observing the same property both in input and in output, i.e., $(\rho_e)\ e\ (\rho_e)$.

[6] Note that, here we don't consider control structures since in the original logic only single variable guards are considered.

[7] $\text{VARS}^{\text{H}}\ (e) \overset{\text{def}}{=} \text{VARS}\ (e) \cap \text{H}$

[8] This is the only rule which cannot be directly implemented in its original formulation.

8. Conclusions

The aim of the present paper is to provide a deeper insight on the strong relation between slicing and dependency. A new point of view is provided on the relation between the standard definition and the computation of slicing, which allows to look at slicing parametrically on the notion of dependency. Since dependency is the notion formalizing what we mean for *relevant* when defining slicing, it is possible to get a general framework where we can obtain new and, possibly, weaker definitions of slicing, deriving from new notions of dependency. By moving from syntactic to semantic dependency, and from concrete to abstract semantic dependency, we move from standard towards semantic slicing, and further towards abstract slicing. Unfortunately, as underlined before, the generalization of the standard approaches to slicing in this abstract dependency framework is not sufficient for generating precise abstract slices, i.e., slices that cut "all" the irrelevant statements w.r.t. an abstract criterion. This is due to the implicit independencies that can be generated in the semantics of control statements, which need more specific analyses for being detected. In this direction can be exploited the strong relation between dependencies and non-interference, since the abstract framework for non-interference has been widely explored.

Another important issue, is that while the syntactic approach to slicing is clearly implementable, the same does not hold, in general, at the semantic level. In this direction, a constructive approach to the characterization of (abstract) semantic dependencies is shown, but still a lot of work has to be done in order to obtain a real implementation. Moreover, the introduction of abstract properties in the context of dependencies leads to discover new interesting points of view. In particular, given a program, a criterion and a fixed set of variables X, we could wonder which is the most precise property which makes the criterion independent from X. The paper shows a constructive approach for dealing with this new challenge. Finally, the new abstract approach to dependencies, is used in the context of language-based security. Due to the strong relation between the notion of non-interference in language-based security and the notion of dependency [6], the constructive approach to abstract dependencies can be exploited in order to approximate, given a program, the strongest harmless attacker, i.e., the most precise observation of the program which is unable to disclose confidential information.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Regional Government of Madrid under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. POPL*. ACM, 1999.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proc. POPL*. ACM, 2006.

[3] T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming*, 64(1), 2007.

[4] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43, 1996.

[5] I. Cartwright and M. Felleisen. The semantics of program dependence. In *Proc. PLDI*. ACM, 1989.

[6] E. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating System Review*, 11(5), 1977.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL*. ACM, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. POPL*. ACM, 1979.

[9] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. POPL*. ACM, 2004.

[10] R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In *Proc. EACSL*, volume 3210 of *LNCS*. SV, 2004.

[11] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2), 2000.

[12] J. Goguen and J. Meseguer. Security policies and security models. In *Proc. SSP*. IEEE, 1982.

[13] H. Hong, I. Lee, and O. Sokolsky. Abstract Slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Proc. SCAM*. IEEE, 2005.

[14] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM TOPLAS*, 11(3), 1989.

[15] S. Hunt and I. Mastroeni. The PER model of abstract non-interference. In *Proc. SAS*, volume 3672 of *LNCS*. SV, 2005.

[16] T. Reps. Algebraic properties of program integration. *Science of Computer Programming*, 17, 1991.

[17] T. Reps and W. Yang. The semantics of program slicing and program integration. In *Proc. Colloq. on Current Issues in Programming Languages*, volume 352 of *LNCS*. SV, 1989.

[18] X. Rival. Abstract dependences for alarm diagnosis. In *Proc. APLAS*, volume 3780 of *LNCS*. SV, 2005.

[19] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[20] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3, 1995.

[21] M. Ward and H. Zedan. Slicing as a program transformation. *ACM TOPLAS*, 29(2), 2007.

[22] M. Weiser. Program slicing. In *Proc. ICSE*. IEEE, 1981.

[23] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.