A

Abstract Program Slicing: an Abstract Interpretation-based approach to Program Slicing

ISABELLA MASTROENI, Università di Verona, Italy
and DAMIANO ZANARDINI, Technical University of Madrid (UPM), Spain

In the present paper we formally define the notion of *abstract program slicing*, a general form of program slicing where properties of data are considered instead of their exact value. This approach is applied to a language with numeric and reference values, and relies on the notion of *abstract dependencies* between program statements.

The different forms of (backward) abstract slicing are added to an existing formal framework where traditional, non-abstract forms of slicing could be compared. The extended framework allows us to appreciate that abstract slicing is a generalization of traditional slicing, since each form of traditional slicing (dealing with syntactic dependencies) is generalized by a semantic (non-abstract) form of slicing, which is actually equivalent to an abstract form where the *identity* abstraction is performed on data.

Sound algorithms for computing abstract dependencies and a systematic characterization of program slices are provided, which rely on the notion of *agreement* between program states.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Formal methods; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Logics of programs; Mechanical verification; F.3.2 [**Semantics of Programming Languages**]: Program analysis; F.4.1 [**Mathematical Logic**]: Computational logic; I.2.2 [**Automatic Programming**]: Program verification

CCS Concepts: •**Security and privacy** → **Logic and verification;** •**Theory of computation** → **Logic and verification; Program verification; Program analysis;** *Automated reasoning; Programming logic;* Hoare logic; Pre- and post-conditions; •**Software and its engineering** → **Formal software verification;**

General Terms: Theory, Analysis, Verification

Additional Key Words and Phrases: Program Slicing, Semantics, Static Analysis, Abstract Interpretation

ACM Reference Format:
Isabella Mastroeni and Damiano Zanardini and Samir Genaim. XXXX. Abstract Program Slicing: an Abstract Interpretation-based approach to Program Slicing. *ACM Trans. Comput. Logic* V, N, Article A (January YYYY), 56 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

It is well-known that, as the size of programs increases, it becomes impractical to maintain them as monolithic structures. Indeed, splitting programs into smaller pieces allows to construct, understand and maintain large programs much more easily. *Program slicing* [Weiser 1984; Tip 1995; Binkley and Gallagher 1996; De Lucia 2001] is a program-manipulation technique that extracts, from programs, those statements which are *relevant* to a particular computation. In the most traditional definition, a *program slice* is an executable program

Authors' addresses: Isabella Mastroeni, Dipartimento di Informatica, Facoltà di Scienze, Università di Verona, Strada Le Grazie 15, 37134 Verona, Italy; Damiano Zanardini, Departamento de Inteligencia Artificial, Escuela Técnica Superior de Ingenieros Informáticos, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain.

DOI: http://dx.doi.org/10.1145/0000000.0000000

```
1 a:=1;                    1 a:=1;                    1                          1
2 b:=b+1;                  2 b:=b+1;                  2 b:=b+1;                  2 b:=b+1;
3 c:=c+2;                  3 c:=c+2;                  3 c:=c+2;                  3
4 e:=e+1;                  4                          4                          4
5 d:=2*c+b+a-a;            5 d:=2*c+b+a-a;            5 d:=2*c+b+a-a;            5 d:=2*c+b+a-a;
```

Program P | Program Q | Program R | Program S

Fig. 1. Q, R and S are, respectively, a *slice*, a *semantic slice* and an *abstract slice* of P.

whose behavior must be identical to a specific subset of the original program's behavior. The specification of this subset is called the *slicing criterion*, and can be expressed as the value of some set of variables at some set of statements and/or program points [Weiser 1984]. Slicing[1] can be and is used in several areas like debugging [Weiser 1984], software maintenance [Gallagher and Lyle 1991], comprehension [Canfora et al. 1998; Field et al. 1995], or re-engineering [Cimitile et al. 1996].

Since the seminal paper introducing slicing [Weiser 1984], there have been many works proposing several notions of slicing, and different algorithms to compute slices (see [Tip 1995; De Lucia 2001] for good surveys about existing slicing techniques). Program slicing is a transformation technique that reduces the size of programs to analyze. Nevertheless, the reduction obtained by means of standard slicing techniques may be not sufficient for simplifying program analyses since it may keep more statements than those strictly necessary for the desired analysis. Suppose we are analyzing a program, and suppose we want a variable x to have a particular property ρ at a given program point n. If we realize that x does not have that property at n, then we may want to understand which statements affect that property of x, in order to find out more easily where the computation went wrong. In this case, we are not interested in the exact value of x, so that we may not need *all* the statements that a standard slicing algorithm would return. Instead, we would need a technique that returns the minimal amount of statements that actually affect that specific property of x.

1.1. Abstract program slicing

This paper introduces and discusses a "semantic" general notion of slicing, called *abstract program slicing*, looking for those statements affecting a property (modeled in the context of abstract interpretation [Cousot and Cousot 1977]) of a set of variables of interest, the so called *abstract criterion*. The idea behind this new notion of slicing is investigating more *semantically* precise notions of dependency between variables. In other words, when a syntactic dependency is detected, such as the dependency, in an assignment, of *defined* variables from *used* variables, we look further for semantic dependencies, i.e., dependencies between *values* of variables.

Consider the program P in Fig. 1, and suppose that we are interested in the variable `d` at the end of the execution. Standard slicing algorithms extract slices by computing syntactic dependencies; in this sense, `d` depends on all `c`, `b` and `a`, so that a sound slice would have to take all the statements involving all these variables. In the figure, Q is a *slice* of P with respect to that criterion. However, if we are interested in a more precise, semantic notion of slicing, then we could observe that the value of `d` only depends on the values of variables `c` and `b`, so that a more precise slice would be represented by R. Finally, if we are interested in the parity of `d` at that point, then we observe that parity of `d` does not depend on the value of `c`, and S is an *abstract slice* of P with respect to the specified criterion. Even in this simple case, the *abstract slice* gives more precise information about the statements affecting the property of interest.

[1] We use *slicing* (*slice*) and *program slicing* (*program slice*) as interchangeable terms.

1.2. Outline and contributions

In this paper, we aim at introducing a generalized notion of slicing, allowing us to weaken the notion of "dependency" (from syntax, to semantics, to abstract semantics) with respect to what is considered *relevant* for computing the slice. Since our generalization is a semantic one, we start from the unifying framework by Binkley *et al.* [Binkley et al. 2006a; 2006b], recalled and discussed in Section 3 and the Appendix, where different forms of slicing are defined and compared w.r.t. their characteristics (static/dynamic, iteration-count/non-iteration-count, etc.), into a comprehensive formal framework. The structure of this framework is based on the formal definition of the criterion, inducing a semantic equivalence relation $\mathcal{E}$ which uniquely characterizes the set of possible slices of a program P as the set of all the sub-programs[2] equivalent to P with respect to $\mathcal{E}$. This structure makes the framework suitable for the introduction and the formal definition of an abstract form of slicing in Section 4, since abstraction corresponds simply to consider a weaker criterion, which implies weakening the equivalence relation $\mathcal{E}$ defining slicing.

Once we have the equivalence relation defining a desired notion of slicing w.r.t. a given criterion, we show in Section 5 how this corresponds to fixing the notion of dependency we are interested in (namely, the notion of dependency determining what has to be considered relevant in the construction of slicing), and we show how the extension to semantic dependencies may be used to extend the program dependency graph-based approach to computing slices [Horwitz et al. 1989]. Finally, we define in Section 6 a notion of abstract dependencies implying abstract criteria. In Section 6.1, we show that this new notion of dependency is not suitable for computing slices by using Program Dependency Graphs, and in Section 6.2 propose the findNdeps algorithm for computing (abstract) dependencies, and the Edep algorithm for finding an abstract domain for which some specific abstract dependency of an expression on some variables does not hold.

Next, Section 7 gives a systematic approach to compute backward slices. Such an approach relies on two systems of logical rules in order to prove (1) Hoare-style tuples capturing the effect of executing a statement s on a pair of states for which some similarity (*agreement*) is required by the slicing criterion (indeed, this similarity corresponds to the semantic equivalent relation); and (2) when some properties of the state do not change (are *preserved*) after s is executed. The combination of the results provided by these rule systems allows to decide whether it is safe to remove a statement from a program without changing the observation corresponding to the criterion. Importantly, the rule systems and algorithms provided in Section 7 rely on the knowledge and manipulation of a "library" of abstract properties. For example, in order to infer that 2∗x is always even, the abstract domain representing the parity of number must be known and available as a "component" of the logical systems. If no abstract property is known except the identity (which is the most precise property, and is not really abstract), then the approach boils down to standard slicing. Importantly, it becomes clear in this case that slices on the same variables (properties of them in the abstract case; exact values in the concrete case) are generally bigger in the concrete setting (when identity is the only available property) than in the corresponding abstract slicing. Needless to say, this does not mean that every algorithm for abstract slicing will perform better than any algorithm for non-abstract slicing; rather, it provides a practical insight of how optimal (purely semantic-based) abstract slices may not include statements which are included in concrete slices.

Part of this work has been previously published in conference proceedings [Mastroeni and Zanardini 2008; Zanardini 2008; Mastroeni and Nikolić 2010]. The present paper joins these works into a coherent framework, and contains a number of novel contributions:

[2]The framework proposed in [Binkley et al. 2006a; 2006b] is parametric on the syntactic relation, but here we only consider the relation of being a subprogram.

- In Section 4, we formally prove that abstract slicing can be put in the formal framework of [Binkley et al. 2006a] and generalizes concrete forms of slicing.
- In the same Section, we formally define the notion of dependency induced by a particular criterion, i.e., by the equivalence relation among programs corresponding to the chosen criterion.
- In Section 5, we define and prove how we can approximate this (concrete semantic) dependency in order to use it for pruning PDGs and computing slicing with the well known PDG-based algorithm for slicing [Reps and Yang 1989].
- In Section 6.1, we discuss why the idea of pruning PDGs is not applicable to the abstract notion of dependency, thus showing the need of providing different approaches for computing abstract slices.
- The discussion on the findNdeps and Edep algorithm in Section 6.2.1 and 6.2.2 has been significantly improved, and several examples have been provided.
- The treatment of non-numerical values when computing slices was already considered in [Zanardini 2008]. However, the language under study in the present paper is different in that it is closer to standard object-oriented languages. More concretely, that work used complex identifiers $x.f.g$ as if they were normal variables, thus obtaining that sharing between variables was easier to deal with. However, this came at the cost of increasing the number of "variables" to be tracked by the analysis. Section 7 discusses the logical machinery to compute backwards slices in this context. Moreover, examples have been provided to illustrate how properties of the heap can be taken into account.
- The G-system introduced in Section 7.1 is a quite refined version of the A-system [Zanardini 2008]; rules for variable assignment and field update have been changed according to the new language (which implies a number of technical issues); there is a new rule G-ID; the overall discussion has been improved.
- The rule system for proving the preservation of properties (the PP-system) is explicitly introduced in Section 7.1.7.
- The description of how statements can be erased has been greatly improved; an algorithm has been explicitly introduced in Section 7.3, which labels each program point with agreements according to the G-system. A thorough discussion and proofs are provided, so that it is guaranteed that the conditions for erasing a statement (relying on the G-system, the PP-system, and the labelSequence procedure for labeling program points with agreements) are sound.
- In Section 7.4.2, recent work on field-sensitive sharing analysis [Zanardini 2015] is included in the computation of abstract slices, which results in improving the precision when data structure in the heap overlap.

2. PRELIMINARIES

2.1. The programming language

The language is a simple imperative language with basic object-oriented features, whose syntax will be easy to understand for anyone who is familiar with imperative programming and object orientation. The language syntax includes the usual arithmetic expressions EXP and access to object fields via "dot" selectors. A statement can be **skip**, a variable assignment x:=e, a field update x.f:=e, a conditional or a **while** loop. In addition, there exist special statements (1) **read** which reads the value of some variable from the input, simulating the use of parameters; this kind of statement can only appear at the beginning of the program; and (2) write, which can only appear at the end of the program and outputs the current value of some variables[3]. For simplicity, guards in conditionals and loops are supposed not to have *side effects*. We denote by $\mathbb{P}$ the set of all programs.

[3]As a matter of fact, this kind of statement is only included for back-compatibility and readability.

$\mathbb{X}$ is the set of program variables and $\mathbb{V}$ denotes the set of values, which can be either integer or reference values, or the **null** constant ($\mathbb{V} = \mathbb{Z} \cup \mathbb{R} \cup \{\mathbf{null}\}$); every variable is supposed to be well-typed (as integer or reference) at every program point. $\mathbb{L}$ denotes the set of *line numbers* (program points). Let $l \in \mathbb{L}$, and $Stm(l)$ be the statement at program line l. For a given program P, we denote by $\mathbb{L}_P \subseteq \mathbb{L}$ the set of all and only the line numbers corresponding to statements of the program P, i.e., $\mathbb{L}_P = \{\ l \in \mathbb{L} \mid Stm(l) \in P\ \}$. This definition is necessary since when we look for slicing we erase statements without changing the numeration of line numbers; for instance, in Figure 1, we have that $Stm(4) \notin Q$, so that $\mathbb{L}_Q = \{1, 2, 3, 5\}$.

A *program state* $\sigma \in \Sigma$ is a pair $\langle n^k, \mu \rangle$ where n is the executed program point, k is the number of times the statement at n has been reached so far, μ is the memory. A *memory* is a pair (ε, h) where the *store* $\varepsilon : \mathbb{X} \to \mathbb{V}$ maps variables to values, and the *heap* h is a sequence of locations where objects can be stored; a reference value corresponds to one of such locations. An *object* o maps field identifiers to values, in the usual way; $o.f$ is the value corresponding to the field f of the object o, and can be either a number, the location in which another object is stored, or **null**. For the sake of simplicity, *classes* are supposed to be declared somewhere, and field accesses are supposed to be consistent with class declarations.

Unless ambiguity may arise, a memory (or even an entire program state) can be considered directly as a store, so that $\mu(x)$ (resp., $\sigma(x)$) will be the value of x in the store contained in μ (resp., in σ). Moreover, a store ε can be represented as $\{x_1 \leftarrow v_1, .., x_m \leftarrow v_m\}$, meaning that $\varepsilon(x_i) = v_i$ for every i, and, again, $\mu = \{x_1 \leftarrow v_1, .., x_m \leftarrow v_m\}$ (resp., $\sigma = \{x_1 \leftarrow v_1, .., x_m \leftarrow v_m\}$) can be used instead of $\varepsilon = \{x_1 \leftarrow v_1, .., x_m \leftarrow v_m\}$ whenever the store is the only relevant part of the memory (resp., the state).

A *state trajectory* $\tau \in \mathbb{T} = \Sigma^*$ is a sequence of program states through which a program goes during the execution. State trajectories are actually traces equipped with the $\overline{k}$ component. The state trajectory obtained by executing program P from the input memory μ is denoted τ_P^μ. Moreover, $\tau[n]$ will be the set of states in τ where the program point is n. Any initial state has $n = 1$, i.e., the set of initial states is $\Sigma_\iota = \{\ \langle 1^1, \mu \rangle \mid \mu \in \mathbb{M}\ \}$.

In the following, $[\![\cdot]\!] : \mathbb{P} \times \wp(\Sigma_\iota) \to \wp(\mathbb{T})$ denotes the program semantics where $[\![P]\!](S)$ returns the set of state trajectories obtained by executing the program P starting from any initial state in $S \subseteq \Sigma_\iota$, i.e., $[\![P]\!](S) = \{\ \tau_P^\mu \mid \langle 1^1, \mu \rangle \in S\ \}$. We abuse notation by denoting in the same way also the semantics of expressions, namely, $[\![\cdot]\!] : \textsc{Exp} \times \Sigma \to \mathbb{V}$, which is such that $[\![e]\!](\sigma)$ ($e \in \textsc{Exp}$) returns the evaluation of e in σ. Finally, if $S \subseteq \Sigma$, in sake of simplicity, we still abuse notation by denoting in the same way also the additive lift of semantics, i.e., $[\![e]\!](S) = \{\ [\![e]\!](\sigma) \mid \sigma \in S\ \}$.

2.2. Basic Abstract Interpretation

This section introduces the lattice of *abstract interpretations* [Cousot and Cousot 1977]. Let $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ denote a complete lattice C, with ordering $\leq$, lub $\vee$, glb $\wedge$, top and bottom element $\top$ and $\bot$, respectively. A *Galois connection* (G.c.) is a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ such that $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. In standard terminology, C and A are, respectively, the concrete and the abstract domain. Abstract domains can be formulated as upper closure operators (ρ) [Cousot and Cousot 1977]. Given an ordered set C with ordering $\leq_C$, a uco on C, $\rho : C \to C$, is a monotone, idempotent ($\rho(\rho(x)) = \rho(x)$) and extensive ($\forall x \in C.\ x \leq_C \rho(x)$) map. Each uco ρ is uniquely determined by the set of its fix-points, which is its image; i.e., $\rho(C) = \{x \in C \mid \rho(x) = x\}$. When $C = \wp(D)$ for some set D, and $v \in D$ then we usually write $\rho(v)$ instead of $\rho(\{v\})$ (and in general for any function, $f(v)$ instead of $f(\{v\})$). If C is a complete lattice, then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x \rangle$ is a complete lattice, where $uco(C)$ is the domain of all the upper closure operators on the lattice C; for every two ucos $\rho_1, \rho_2 \in uco(C)$, $\rho_1 \sqsubseteq \rho_2$ if and only if $\forall y \in C.\ \rho_1(y) \leq \rho_2(y)$ iff $\rho_2(C) \subseteq \rho_1(C)$; and, for every $\{\rho_i\}_{i \in I} \subseteq uco(C)$, $(\sqcap_{i \in I} \rho_i)(x) = \wedge_{i \in I} \rho_i(x)$ and

$(\sqcup_{i\in I}\rho_i)(x) = x \Leftrightarrow \forall i \in I.\ \rho_i(x) = x$. In the following we will denote by ρ_{ID} the most concrete uco on a domain, i.e., $\lambda x.x$, and by $\rho_\top$ the most abstract one $\lambda x.\top$. A_1 is more precise than A_2 (i.e., A_2 is an abstraction of A_1) iff $A_1 \sqsubseteq A_2$ in $uco(C)$. The *reduced product* of a family $\{\rho_i\}_{i\in I}$ is $\sqcap_{i\in I}\rho_i$ and is one of the best-known operations for composing domains.

Example 2.1 (*Numerical abstract domains*). Let the concrete domain C be $\wp(\mathbb{Z})$: the *parity* abstract domain ρ_{PAR} in Figure 2 (on the left) represents the parity of numbers, and is determined by fix-points $\{[\mathbf{bot}], [\mathbf{even}], [\mathbf{odd}], [\mathbf{top}]\}$ where [**even**] and [**odd**] denote even and odd numbers, respectively; [**bot**] is the empty set, and [**top**] $= \mathbb{Z}$. For example, $\rho_{\text{PAR}}(\{2,4,10\}) =$ [**even**] (all numbers are even), $\rho_{\text{PAR}}(\{3,7\}) =$ [**odd**] (both numbers are odd), and $\rho_{\text{PAR}}(\{4,5\}) =$ [**top**] (there are both even and odd numbers). The *sign* abstract domain ρ_{SIGN} (in the center of Figure 2) is characterized by fix-points $\{[\mathbf{bot}], [\mathbf{zero}], [\mathbf{pos}], [\mathbf{neg}], [\mathbf{top}]\}$ and tracks the sign of integers (zero, positive, negative, etc.). For example, $\rho_{\text{SIGN}}(\{0\}) =$ [**zero**], $\rho_{\text{SIGN}}(\{-3,-4,-5\}) =$ [**neg**], $\rho_{\text{SIGN}}(\{1,2,4\}) =$ [**pos**], $\rho_{\text{SIGN}}(\{1,-1\}) =$ [**top**]. Finally, the *parity-sign* domain ρ_{PARSIGN} (on the right), which is the reduced product $\sqcap$ of ρ_{PAR} and ρ_{SIGN}, captures both properties (the parity and the sign), and has fix-points [**bot**], [**zero**], [**poseven**], [**posodd**], [**negeven**], [**negodd**], [**even**], [**odd**], [**pos**], [**neg**], and [**top**].

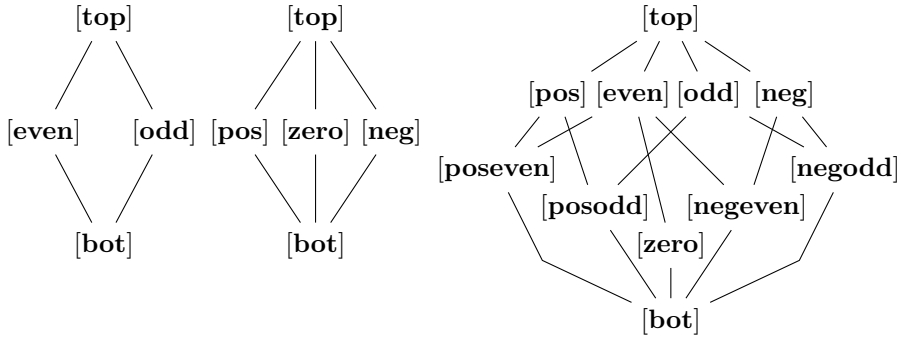


Fig. 2. The ρ_{PAR}, ρ_{SIGN} and ρ_{PARSIGN} domains.

Formally speaking, the value of a reference variable is either a location ℓ or **null**. However, the domains introduced in the next example classify variables not only with respect to ℓ itself, but also on the data structure in the heap which is reachable from ℓ. This point of view is similar to previous work on static analysis of properties of the heap like *sharing* [Secci and Spoto 2005] or *cyclicity* [Rossignoli and Spoto 2006; Genaim and Zanardini 2013].

Example 2.2 (*Reference abstract domains*). Let C be $\wp(\mathbb{R}\cup\{\mathbf{null}\})$, i.e., the possible values of reference variables. The *nullity* domain ρ_{NULL} classifies values on nullity, and has fix-points $\{[\mathbf{bot}], [\mathbf{null}], [\mathbf{non{-}null}], [\mathbf{top}]\}$ where the concretizations of [**null**] and [**non−null**] are, respectively, {**null**} and $\mathbb{R}$.

On the other hand, it is possible to define a *cyclicity* domain ρ_{CYC} which classifies variables on whether they point to *cyclic* or *acyclic* data structures [Genaim and Zanardini 2013]. A *cycle* in the heap is a path in which the same location is reached more than once; a double-linked list (one which can be traversed in both directions) is a good example of a cyclic data structure. The fix-points of this domain are $\{[\mathbf{bot}], [\mathbf{cyc}], [\mathbf{acyc}], [\mathbf{top}]\}$, where all acyclic values (including **null**) are abstracted to [**acyc**], and all cyclic values (i.e., locations from which a cycle is reachable) are abstracted to [**cyc**]. Both domains and their reduced product

are depicted in Figure 3; note that there are no values which are both null and cyclic, so that their intersection collapses to [**bot**].

Finally, the identity domain ρ_{ID}, abstracts two concrete values to the same abstract value only if they are equal. Two references are *equal* if (1) their are both null; or (2) they are both non-null and the objects stored in the corresponding locations are equal, where equality on objects means that all their numeric fields must be the same number and all reference fields must be equal (w.r.t. this same notion of equality on references).

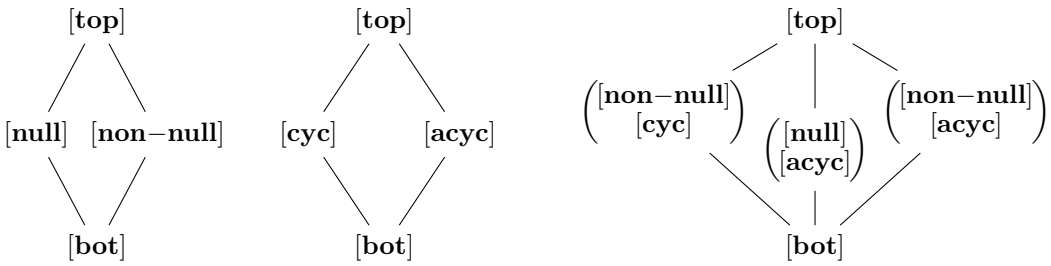


Fig. 3. The ρ_{NULL} and the ρ_{CYC} domains, and their reduced product.

Let us consider now $D = C^n$ (C lattice and $n \in \mathbb{N}$), namely $x \in D$ is a n-tuple of elements of C, and consider $\rho \in uco(D)$. In this case, we can distinguish between two kinds of abstractions: *non-relational* and *relational* abstractions [Cousot 2001; Cousot and Cousot 1979]. The non-relational or *attribute-independent* one [Cousot and Cousot 1979, Example 6.2.0.2] consists in ignoring the possible relationships between the values of the abstracted inputs. For instance, if ρ is applied to the values of variables x and y, then ρ can be approximated through projection by a pair of abstractions on the single variables, analyzing the single variables in isolation. In sake of simplicity, without losing generality, consider $n = 2$, i.e., $D = C^2 = C \times C$. Formally, $\rho \in uco(C \times C)$ is non-relational if there exist $\delta_1, \delta_2 \in uco(C)$ such that $\rho(x, y) = \langle \delta_1(x), \delta_2(y) \rangle$, i.e, $\rho \in uco(C) \times uco(C) \subset uco(C \times C)$. For instance, let ρ_{PAR} be the abstract domain depicted in Figure 2 expressing the parity of integer values; the ρ_{PAR} non-relational property of $\langle x, y \rangle$ provides the parity of x and y independently one from each other, meaning that all the possible combinations of parity of x and y are possible as results ($\langle [\mathbf{even}], [\mathbf{even}] \rangle$, $\langle [\mathbf{even}], [\mathbf{odd}] \rangle$, $\langle [\mathbf{odd}], [\mathbf{even}] \rangle$, $\langle [\mathbf{odd}], [\mathbf{odd}] \rangle$ and all combinations where at least one variable is [**top**] or [**bot**]). Relational abstractions may preserve some of the relationship between the analyzed values [Cousot 2001]. For instance, we could define an abstraction preserving the fact that x is even ([**even**]) if and only if y is odd ([**odd**]). It is clear that, in this case, we are more precise since the only possible analysis results are $\langle [\mathbf{even}], [\mathbf{odd}] \rangle$, $\langle [\mathbf{odd}], [\mathbf{even}] \rangle$, $\langle [\mathbf{top}], [\mathbf{top}] \rangle$ and $\langle [\mathbf{bot}], [\mathbf{bot}] \rangle$.

If $\rho \in uco(C)$, $f \in C \longrightarrow C$, and $f^\sharp \in \rho(C) \longrightarrow \rho(C)$, then $f^\sharp$ is a *sound* approximation of f if $\rho \circ f \sqsubseteq f^\sharp \circ \rho$. $f^\rho \stackrel{\text{def}}{=} \rho \circ f \circ \rho$ is known as the *best correct approximation* (bca) of f in ρ, which is always sound by construction. Soundness naturally implies fix-point soundness, that is, $\rho(lfp^{\leq_C}_{\perp_C} f_C) \leq_\rho lfp^{\leq_C}_{\perp_C} f^\rho$. If $\rho \circ f = \rho \circ f \circ \rho$ then we say that f^ρ is a *complete* approximation of f [Cousot and Cousot 1979; Giacobazzi et al. 2000]. In this case, $\rho(lfp^{\leq_C}_{\perp_C} f) = lfp^{\leq_C}_{\perp_C} f^\rho$.

2.3. Equivalence relations, abstractions and partitions

Closure operators and equivalence relations are related concepts [Cousot and Cousot 1979]. Recently, this connection has been further studied in the field of abstract model checking and language based-security [Ranzato and Tapparo 2002; Hunt and Mastroeni 2005]. In

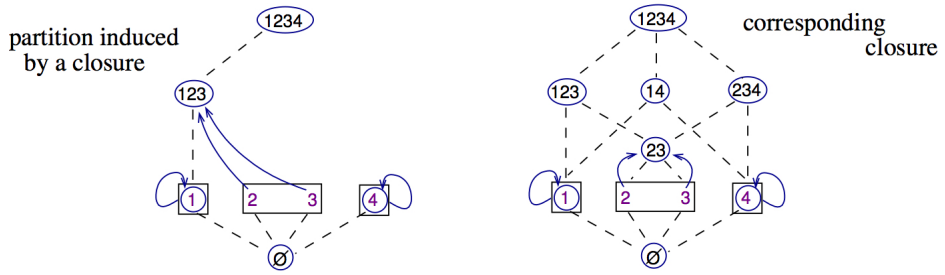


Fig. 4. A partitioning closure.

particular, there exists an isomorphism between equivalence relations and a subclass of upper closure operators. Consider a set S: for each equivalence relation $\mathtt{R} \subseteq S \times S$ we can define an upper closure operator, $Clo^{\mathtt{R}} \in uco(\wp(S))$ such that $\forall x \in S \,.\, Clo^{\mathtt{R}}(\{x\}) = [x]_{\mathtt{R}}$ and $\forall X \subseteq S \,.\, Clo^{\mathtt{R}}(X) = \bigcup_{x \in X} [x]_{\mathtt{R}}$. Conversely, for each upper closure operator $\eta \in uco(\wp(S))$, we are able to define an equivalence relation $Rel^{\eta} \subseteq S \times S$ such that $\forall x, y \in S \,.\, x \; Rel^{\eta} \; y \Leftrightarrow \eta(\{x\}) = \eta(\{y\})$. It is immediate to prove that Rel^{η} is an equivalence relation, and this comes from η being merely a function, not necessarily a closure operator. $Clo^{\mathtt{R}}$ is identified as the most concrete closure η such that $\mathtt{R} = Rel^{\eta}$ [Hunt and Mastroeni 2005]. It is possible to associate with each upper closure operator the most concrete closure inducing the same partition on the concrete domain S:

$$\Pi(\eta) \stackrel{\text{def}}{=} Clo^{Rel^{\eta}} \tag{1}$$

Note that, for all $\eta \in uco(\wp(S))$, $\Pi(\eta)$ is the (unique) most concrete closure that induces the same equivalence relation as η ($Rel^{\eta} = Rel^{\Pi(\eta)}$). The fix-points of Π are called the *partitioning* closures. Being $\wp(S)$ a complete Boolean lattice, an upper closure operator $\eta \in uco(\wp(S))$ is partitioning, i.e., $\eta = \Pi(\eta)$, iff it is complemented, namely if $\forall X \in \eta. \, \overline{X} \stackrel{\text{def}}{=} S \setminus X \in \eta$ [Hunt and Mastroeni 2005].

Example 2.3. Consider the set $S = \{1, 2, 3, 4\}$ and one of its possible partitions $\pi = \{\{1\}, \{2, 3\}, \{4\}\}$. The closure η with fix-points $\{\emptyset, \{1\}, \{4\}, \{1, 2, 3\}, S\}$ induces exactly π as a state partition, but the most *concrete* closure that induces π is $Clo^{\pi} = \Pi(\eta) = \bigvee(\{\emptyset, \{1\}, \{2, 3\}, \{4\}\}, S)$, which is the closure on the right of Figure 4.

Given a partitioning upper closure operator ρ, an *atom* is an element a of ρ such that there does not exists another element b with $[\mathbf{bot}] \sqsubset b \sqsubset a$. For example, the atoms of ρ_{PARSIGN} are [**poseven**], [**posodd**], [**zero**], [**negeven**], and [**negodd**]. In partitioning closures, atoms are all the possible abstractions of singletons: in fact, $\rho_{\text{PARSIGN}}(\{n\})$ will never give [**pos**] or [**odd**] since there is always a more precise abstract value describing n. In the following, $\text{ATOM}_{\rho}(a)$ holds iff a is an atom of ρ.

2.4. Abstract semantics and static analysis

An abstract program semantics is the abstract counterpart of the concrete semantics w.r.t. an abstract program observation: it is meant to compute, for each program point, an abstract state which soundly represents *invariant properties* of variables at that point. In general, it is computed by an abstract interpreter [Cousot and Cousot 1979] collecting the set of all the possible values that each variable may have in each program point and abstracting this set in the chosen abstract domain.

Given a concrete program state σ and an abstract domain $\rho \in uco(\wp(\mathbb{V}))$, an *abstract state* $\sigma^{\rho} \in \Sigma^{\rho}$ is obtained by applying the abstraction ρ to the values of variables stored in it. Namely, $\sigma^{\rho} = \langle n^k, \mu^{\rho} \rangle$, where $\mu^{\rho} = \langle \varepsilon^{\rho} \rangle$ and ε^{ρ} is such that $\varepsilon^{\rho}(x) = \rho(\varepsilon(x))$. For simplicity,

we can write $\sigma^\rho(x) = \rho(\sigma(x))$, treating the whole state as a store when applied to variables. In the case of a reference variable x, the abstraction $\sigma^\rho(x)$ gives information about the data structure pointed to by x (e.g., if $\rho = \rho_{\text{CYC}}$, the cyclicity of the data structure can be represented). This explains why the heap is not represented explicitly in the abstract state: instead, relevant information about the heap is contained in the abstraction of variables (see the previous discussion before Example 2.2).

In the following, ordering $\leq$ on abstract states is variable-wise comparison between abstract values:

$$\sigma^\rho{}_1 \leq \sigma^\rho{}_2 \quad \Leftrightarrow \quad \forall x. \sigma^\rho{}_1(x) \subseteq \sigma^\rho{}_2(x)$$

The greater an abstract state is, the wider is the set of concrete states it represents; if $\sigma^\rho{}_1 \leq \sigma^\rho{}_2$, then $\sigma^\rho{}_1$ is called a *refinement* of $\sigma^\rho{}_2$. Moreover, a *covering* of σ^ρ is a set of refinements $\{\sigma^\rho{}_1 .. \sigma^\rho{}_n\}$ such that $\vee_i \sigma^\rho{}_i = \sigma^\rho$. The set of abstract state trajectories is $\mathbb{T}^{\stackrel{\leq}{\approx}} = \Sigma^{\rho *}$, i.e., an abstract trajectory is the computation of a program on the set of abstract states. The trace in $\mathbb{T}^{\stackrel{\leq}{\approx}}$ of a program P, starting from the abstract memory μ^ρ is denoted by $\tau_P^{\mu^\rho}$.

The *abstract program semantics* $[\![\cdot]\!]^\rho : \mathbb{P} \times \Sigma_\iota^\rho \to \mathbb{T}^{\stackrel{\leq}{\approx}}$ is such that $[\![P]\!]^\rho(S) = \{\tau_P^{\mu^\rho} \mid \langle 1^1, \mu^\rho \rangle \in S\}$ is the set of the sequences of abstract states computed starting from the abstract initial states in $S \in \wp(\Sigma_\iota^\rho)$. We also abuse notation by denoting $[\![\cdot]\!]^\rho$ also the abstract evaluation of expressions. Namely, $[\![\cdot]\!]^\rho : \mathbb{E} \times \Sigma^\rho \to \rho(\mathbb{V})$ is such that $\forall x. [\![e]\!]^\rho(\sigma^\rho(x)) = \rho([\![e]\!](\sigma^\rho(x))) = \rho([\![e]\!](\rho(\sigma(x))))$. This definition is correct, since by construction, we have that any abstract state σ^ρ corresponds to a set of concrete states, i.e., $\sigma^\rho = \{\ \overline{\sigma} \in \Sigma\ \mid \overline{\sigma}^\rho = \sigma^\rho\ \} = \{\ \overline{\sigma} \in \Sigma\ \mid \forall x \in \mathbb{X}. \overline{\sigma}^\rho(x) = \sigma^\rho(x)\ \}$, namely, it is the set of all the concrete states having as abstraction in ρ precisely σ^ρ, and we abuse notation by denoting with $[\![\cdot]\!]^\rho$ also its additive lift. In other words, $[\![e]\!]^\rho$ is the best correct approximation of $[\![e]\!]$ by means of an abstract value in ρ. In general, in order to compute the abstract semantics of a program on an abstract domain ρ, we have to equip the domain ρ with the abstract versions of all the operators used for defining expressions. In our language, we should define, for example, the meaning of $+$, $-$, $*$ and $/$ on abstract values, i.e., on sets of concrete values. This is standard in abstract interpretation, and these operations are defined for many known numerical abstract domains. For instance, the sound approximation of the sum operation on ρ_{PAR} is the following:

$$\begin{array}{lll} [\textbf{even}] + [\textbf{even}] = [\textbf{even}] & \qquad & [\textbf{top}] + \_ = [\textbf{top}] \\ [\textbf{even}] + [\textbf{odd}] = [\textbf{odd}] & & \_ + [\textbf{top}] = [\textbf{top}] \\ [\textbf{odd}] + [\textbf{even}] = [\textbf{odd}] & & [\textbf{bot}] + \_ = [\textbf{bot}] \\ [\textbf{odd}] + [\textbf{odd}] = [\textbf{even}] & & \_ + [\textbf{bot}] = [\textbf{bot}] \end{array}$$

We can reason similarly for all the other operators.

In the following, the notation $[\![e]\!]^\rho(S)$ where S is a set of concrete states is used to denote the abstract evaluation of e in an abstract state which is obtained by abstracting S; as usual, $[\![e]\!]^\rho(\sigma)$ stands for $[\![e]\!]^\rho(\{\sigma\})$. The use of $[\![\cdot]\!]^\rho$ in Section 6.2.1 and later in the paper is twofold: (1) to infer invariant properties, as in Example 2.4; and (2) to evaluate expressions at the abstract level.

Example 2.4. Consider the following code fragment:

```
1 i := 10;
2 j := 0;
3 while (i≥0) {
4     i := i−1;
5     j := j+1;
```

```
6 }
```

and an abstraction $\rho = \rho_{\text{SIGN}}$, i.e., the property of interest is the sign of both i and j. By computing the abstract semantics of this simple program, we can observe that inside the loop we lose the sign of i since i starts being positive, but then the $\mathsf{i} - 1$ operation makes impossible to know statically the sign of i (the result may be positive, zero or negative starting from i positive or zero), while we have that j always remains positive. Moreover, if the loop terminates we can surely say that, at the end, $\mathsf{i} < 0$, namely it is negative (due to the negation of the while guard). Hence, we are able to infer that i is negative and j is positive after line 6. This means that the final abstract state σ^ρ is such that $\sigma^\rho(\mathsf{i}) = [\mathbf{neg}]$ and $\sigma^\rho(\mathsf{j}) = [\mathbf{pos}]$ (in the following, the extensional notation for σ^ρ will be $\{\mathsf{i} \leftarrow [\mathbf{neg}], \mathsf{j} \leftarrow [\mathbf{pos}]\}$, similar to the notation for concrete states).

3. PROGRAM SLICING

Program slicing [Weiser 1984] is a program-manipulation technique which extracts from programs those statements which are relevant to a particular portion of a computation. In order to answer the question about which are the relevant statements, an observer needs a *window* through which only a part of the computation can be seen [Binkley and Gallagher 1996]. Usually, what identifies the portion of interest in the computation is the value of some set of variables at a certain program point, so that a *program slice* comes to be the subset (syntactically, in terms of statements) of the original program which contributes directly or indirectly to the values assumed by some set of variables at the program point of interest. The *slicing criterion* is what specifies the part of the computation which is relevant to the analysis; in this case, a criterion is a pair consisting of a set $\mathcal{X}$ of variables and a program point (or line number) n. The following definition [Binkley and Gallagher 1996] is a possible formalization of the original idea of program slicing [Weiser 1984], in the case of a single variable:

DEFINITION 3.1. *[Binkley and Gallagher 1996] For a statement s (at program point n) and a variable x, the 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, it is also reached in P', and 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.

It is worth noting that Reps and Yang [Reps and Yang 1989], in their *slicing theorem*, provide implicitly a similar definition of program slicing, but it only considers terminating computations. The following example provides the intuition of how slicing works.

Example 3.2. Consider the word-count program [Majumdar et al. 2007] given in Figure 5. It takes in a block of text and outputs the number of lines (nl), words (nw) and characters (nc). Suppose the slicing criterion only cares for the value of nl at the end of the program; then a possible slice is on the left in Figure 6. On the other hand, if the criterion is only interested in nw, then a correct slice is on the right.

Starting from the original definition [Weiser 1984], the notion of slicing has gone through several generalizations and versions, but one feature is constantly present: the fact that slicing is based on a notion of *semantic equivalence* that has to hold between a program and its slices or on a corresponding notion of *dependency*, determining what we keep in the slice while preserving the equivalence relation. What we can observe about definitions of slicing such as the one given in Definition 3.1 is that they are enough precise for finding algorithms

```
1  int c, nl := 0, nw := 0, nc := 0;
2  int in := false;
3  while ((c=getchar())!=EOF) {
4    nc := nc+1;
5    if (c=' ' || c='\n' || c='\t') {
6      in := false; }
7    elseif (in = false) {
8      in := true;
9      nw := nw+1; }
10   if (c = '\n') {
11     nl := nl+1; }
12 }
```

Fig. 5. Word-count program.

```
1  int c, nl := 0;
2
3  while ((c=getchar())!=EOF) {
4
5
6
7
8
9
10   if (c = '\n') {
11     nl := nl+1; }
12 }
```

```
1  int c, nw := 0;
2  int in := false;
3  while ((c=getchar())!=EOF) {
4
5    if (c=' ' || c='\n' || c='\t') {
6      in := false; }
7    elseif (in = false) {
8      in := true;
9      nw := nw+1; }
10
11
12 }
```

Fig. 6. Slices of the word-count program.

for soundly computing slicing, such as [Reps and Yang 1989], but not enough formal to become suitable to generalizations allowing us to compare different forms of slicing and/or to define new weaker forms of slicing.

In the following, we use the formal framework proposed in [Binkley et al. 2006a] where several notions and forms of slicing are modeled and compared. This is not the only attempt to provide a formal framework for slicing (see Section 8), but we believe that, due to its semantic-based approach, it is suitable to include an abstraction level to slicing, which can be easily compared with all the other forms of slicing included in the original framework. Hence, in the following section we do not rewrite a formal framework; rather, we re-formalize the notion of slicing criterion in order to easily include abstractions as a new parameter. A brief introduction of the formal framework together with some examples showing the differences between the different forms of slicing introduced in the following is given in the Appendix.

3.1. Defining Program Slicing: the formal framework

In this section, our aim is to define the forms of slicing that can be lifted to the abstract level. Namely, we consider an existing framework [Binkley et al. 2006a; 2006b] which allows defining abstract slicing simply by defining an abstract criterion which, independently from the kind of slicing (static, dynamic, conditional, standard, etc.), observes properties instead of concrete (exact) values. To this end, we need to slightly revise the construction in order to provide a completely unified notation for slicing criteria. The present paper will only deal with *backward* slicing, where the interest is on the part of the program which *affects* the observation associated with the slicing criterion and not on the part of the program which *is affected* by such an observation (called instead *forward* slicing [Tip 1995]).

Defining slicing criteria. The slicing criterion characterizes what we have to observe of the program in order to decide whether a program is a slice or not of another program. In particular, we have to fix which computations have to be compared, i.e., the inputs and the observations on which the slice and the program have to agree.

In the seminal Weiser approach, given a set of variables of interest $\mathcal{X}$ and program statement s, here referred by the program point n where s is placed, a slicing criterion was modeled as $C = (\mathcal{X}, n)$. In the following, we will gradually enrich and generalize this model in order to include several different notions and forms of slicing. Weiser's approach is known as *static slicing* since the equivalence between the original program and the slice has, implicitly, to hold for every possible input. On the other hand, Korel and Laski proposed a new technique called *dynamic slicing* [Korel and Laski 1988] which only considers one particular computation, and therefore one particular input, so that the dynamic slice only preserves the (subset of the) meaning of the original program for that input. Hence, in order to characterize a slicing criterion including also dynamic slicing we have to add a parameter describing the set of initial memories $\mathcal{I} \subseteq \mathbb{M}$: The criterion is now $C = (\mathcal{I}, \mathcal{X}, n)$, where $\mathcal{I} = \mathbb{M}$ for static slicing, while $\mathcal{I} = \{\mu\}$, with $\mu \in \mathbb{M}$, for dynamic slicing. Finally, Canfora *et al.* proposed *conditioned slicing* [Canfora et al. 1998], which requires that a conditioned slice preserves the meaning of the original program for a set of inputs satisfying one particular condition φ. Let $\mathcal{I} = \{\ \mu \in \mathbb{M} \mid \mu \text{ satisfies } \varphi\ \}$ be the set of input memories satisfying φ [Binkley et al. 2006a]. Hence, the slicing criterion still can be modeled as $C = (\mathcal{I}, \mathcal{X}, n)$.

Each type of slicing comes in four forms which differ on what the program and the slices must agree on, namely on the observable semantics that has to agree. In the following, we provide an informal definition of these forms in order to provide the intuition of what will be formally defined afterwards:

. *Standard*: It considers one point in a program with respect to a set of variables. In other words, the standard form of slicing only tracks one program point. Semantically, this form of slicing consists in comparing the program and the slices in terms of the (*denotational*) I/O semantics from the program inputs selected by the criterion. Namely, for each selected input, the results of the criterion variables in the point of observation must be the same, independently from the executed statements.
. *Korel and Laski* (`KL`): It is a stronger form where the program and the slice must follow identical paths [Korel and Laski 1988]. Semantically, we could say that the program and the slice must have the same (*operational*) trace semantics w.r.t. the statements kept in the slice, starting from the program inputs selected by the criterion. In other words, as before, the final value must be the same, but in this case these values must be obtained by executing precisely the same statements, i.e., the sequence of statements involved in the execution of the slice is the same as the original one, apart from the fact that removed statements are missing (see an example at the beginning of the Appendix).
. *Iteration count* (`IC`): When considering the trace semantics, the same program point inside a loop may be visited more than once, in the following we call *k-th iteration* of a program point n the k-th time the program point n is visited. The iteration count form of slicing requires that a program and its slice agree only at a particular k-th iteration of a program point of interest. In this way, when a point of interest is inside a loop, we have the possibility to require that the variables must agree only at some iterations of the loop and not always.
. *Korel and Laski iteration count* (`KL`i): It is the combination of the last two forms.

In order to deal with these different forms of slicing, the slicing criterion must be enriched with additional information. In particular, the `KL` form of slicing does not change where to observe variables, but it does change the observed semantics up to that point. Hence, we simply have to add a boolean parameter ψ: *true* means that we are considering a `KL` form and we require that the slice must agree with the program on the execution of statements that

are in the slice (and obviously also in the original program); on the other hand, *false* indicates a standard, non-KL form of slicing. Hence, a criterion C comes to be $(\mathcal{I}, \mathcal{X}, n, \psi)$.

The IC form, instead, affects the observation: in order to embed this features in the criterion, the third parameter has to be changed. Let $\{k_1, .., k_j\} \subseteq \mathbb{N}$ be the iterations of the program point $n \in \mathbb{L}$ we are interested in; then, instead of n, in the third parameter of the criterion we should have $\langle n, \{k_1, .., k_j\}\rangle$. Therefore, C takes the form $(\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi)$, where $\mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N})$. Note that $\langle n, \mathbb{N}\rangle$ represents the fact that we are interested in all occurrences of n, as it happens in the standard form.

There are also some *simultaneous* (SIM) forms of slicing that consider more than one program point of interest. In order to deal with SIM forms of slicing, we simply extend the definition of a slicing criterion by considering $\mathcal{O}$ as a set instead of a singleton, namely, $\mathcal{O} \in \wp(\mathbb{L} \times \wp(\mathbb{N}))$.

In the Appendix there are some simple examples showing the main differences between the several forms of slicing introduced so far.

4. ABSTRACT PROGRAM SLICING

This section defines a weaker notion of slicing based on Abstract Interpretation. In particular, we generalize the existing formal framework discussed in Section 3.1 in order to include abstract versions of slicing.

Program slicing is used for reducing the size of programs to analyze. Nevertheless, this reduction may be insufficient in order to really improve an analysis. Suppose that some variables at some point of execution do not have a desired property (for example, that they are different from 0, or from **null**); in order to understand where the error occurred, it would be useful to find those statements which affect such a property of these variables. Standard slicing may return too many statements, making it hard for the programmer to realize which one caused the error.

Example 4.1. Consider the following program P, that inserts a new element elem at position pos in a single-linked list. For simplicity, let pos never exceed the length of list .

```
34 y := null;
35 x := list;
36 while (pos>0) {
37   y := x;
38   x := x.next; // by hypothesis, this always succeeds
39   pos := pos-1;
40 }
41 z := new Node(elem);
42 z.next := x;
43 if (y = null) {
44   list := z;
45 } else {
46   y.next = z;
47 }
```

Suppose that list is cyclic after line 47, i.e., a traversal of the list visits the same node twice. A close inspection of the code reveals that no cycle is created between lines 34 and 47: list is cyclic after line 47 if and only if it was cyclic before line 34.

In the standard approach, it is possible to set the value of list after line 47 as the slicing criterion. In this case, since list can be modified at lines 41–47, at least this piece of code must be included in the slice.

On the other hand, let the cyclicity of list after line 47 be the property of interest, represented by ρ_{CYC} (Example 2.2). Since this property of list does not change, the entire code can be removed from the slice.

4.1. Defining Abstract Program Slicing

We introduce *abstract program slicing*, which compares a program and its abstract slices by considering *properties* instead of exact values of program variables. Such properties are represented as abstract domains, based on the theory of *Abstract Interpretation* (Section 2.2).

We first introduce the notion of *abstract slicing criterion*, where the property of interest is also specified. For the sake of simplicity, the definition only refers to non-SIM forms (i.e., $\mathcal{O}$ is a singleton instead of a set of occurrences: $\mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N})$). In order to make abstract the criterion, we have to formalize in it the *properties* we want to observe on program variables. In particular, we could think of observing different properties for different variables. We define a criterion abstraction $\mathcal{A}$ defined as a tuple of abstract domains, each one relative to a specific subset of program variables: let $\mathcal{X}$ be a set of variables of interest in P and $\{X_i\}_{i\in[1,k]} \subseteq \wp(\mathcal{X})$ a partition of $\mathcal{X}$, the notation $\mathcal{A} = \langle X_1 : \rho_1, .., X_k : \rho_k\rangle$ means that each uco ρ_i is applied to the set of variable X_i (left implicit when it is clear from the context), meaning that ρ_i is precisely the property to observe on X_i. In the following, we denote by $\mathcal{A}_{|X_i}$ the property observed on X_i, formally $\mathcal{A}_{|X_i} = \rho_i$. This is the most general representation, accounting also for relational domains. When ucos will be applied to singletons, the notation will be simplified ($x : \rho$ instead of $\{x\} : \rho$).

Example 4.2. Let x, y, z and w be the variables in $\mathcal{X}$. Let $\mathcal{A}$ be $\langle x : \rho_{\text{PAR}}, \{y, z\} : \rho^{+}_{\text{INT}}, w : \rho_{\text{SIGN}}\rangle$, meaning that the interest is on the parity of x, the sign of w, and the (relational) property of *intervals* [Cousot and Cousot 1979] of the value $y + z$. When abstracting a criterion w.r.t. $\mathcal{A}$, the required observation at a program state σ is

$$\rho_{\text{PAR}}(\sigma(x)) \qquad \rho^{+}_{\text{INT}}(\sigma(y) + \sigma(z)) \qquad \rho_{\text{SIGN}}(\sigma(w))$$

In order to be as general as possible, we consider *relational* properties of variables (see Section 2), so that properties are associated with tuples instead of single variables. In this case, a property is said to *involve* some set (tuple) of variables. Given a memory μ, $\rho(\mu)$ is the result of applying ρ to the values in μ of the variables involved by the abstract domain, and $\mathcal{A}(\mu)$ is the corresponding notion for tuples of ucos.

Definition 4.3 (Abstract criterion). *Let $\mathcal{I} \subseteq \mathbb{M}$ be a set of input memories, $\mathcal{X} \subseteq \mathbb{X}$ be a set of variables of interest; $\mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N})$ be a set of occurrences of interest; ψ be a truth value indicating if the slicing is in* KL *form. Moreover, let $\mathcal{X}$ be the set of variables of interest and $\mathcal{A} \stackrel{def}{=} \langle X_1 : \rho_1, \ldots, X_k : \rho_k\rangle$, with $\{X_i\}_{i\in[1,k]}$ a partition of $\mathcal{X}$. Then, the* abstract slicing criterion *is $C_{\mathcal{A}} = (\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi, \mathcal{A})$,*

Note that, when dealing with non-abstract notions of slicing, we have that each domains is the identity on each single variable, namely $\mathcal{A} = \langle x_1 : \rho_{\text{ID}}, \ldots, x_k : \rho_{\text{ID}}\rangle$, where $\rho_{\text{ID}} \stackrel{\text{def}}{=} \lambda x.x$. It is also worth pointing out that, exactly as it happens for non-abstract forms, $\mathcal{I} = \mathbb{M}$ corresponds to static slicing, and $|\mathcal{I}| = 1$ corresponds to dynamic slicing; in the intermediate cases, we have conditioned slicing.

4.2. The extended formal framework

In this section, we extend a formal framework in which all forms of abstract slicing can be formally represented. It is an extension of the mathematical structure introduced by Binkley *et al.* [Binkley et al. 2006a]. Following their framework, we represent a form of abstract slicing by a pair $(\sqsubseteq, \mathcal{E}_{\mathcal{A}})$, where $\sqsubseteq$ is the traditional syntactic ordering, and $\mathcal{E}_{\mathcal{A}}$ is a function mapping abstract slicing criteria to semantic equivalence relations on programs.

Given two programs P and Q, and an abstract slicing criterion $C_{\mathcal{A}}$, we say that Q is a $(\sqsubseteq, \mathcal{E}_{\mathcal{A}})$-*(abstract)-slice* of P with respect to $C_{\mathcal{A}}$ iff $Q \sqsubseteq P$ and $\langle P, Q\rangle \in \mathcal{E}_{\mathcal{A}}(C_{\mathcal{A}})$ (i.e., P and Q are equivalent w.r.t. $\mathcal{E}_{\mathcal{A}}$). Some preliminary notions are needed to define $\mathcal{E}_{\mathcal{A}}$ in the context of abstract slicing.

An *abstract memory* w.r.t. a set of variables of interest $\mathcal{X}$ (partitioned in $\{X_i\}_{i\in[1,k]}$) is obtained from a memory by restricting its domain to $\mathcal{X}$, and assigning to each set X_i of variables an abstract value determined by the corresponding abstract property of interest ρ_i.

Definition 4.4. *Let $\mu \in \mathbb{M}$ be a memory, $\mathcal{X}$ be the set of a tuple of sets of variables of interest, and $\mathcal{A} = \langle X_1 : \rho_1, \ldots, X_k : \rho_k\rangle$ be the corresponding tuple of properties of interest such that $\{X_i\}_{i\in[1,k]}$ is a partition of $\mathcal{X}$. The* abstract restriction *of a memory μ w.r.t. the state abstraction $\mathcal{A}$ is defined as $\mu \restriction^{\alpha}_{\mathcal{A}} \mathcal{X} \stackrel{def}{=} \mathcal{A} \circ \mu(\mathcal{X}) \stackrel{def}{=} \langle \rho_1(\mu(X_1)), \ldots, \rho_k(\mu(X_k))\rangle$.*

Example 4.5. Let $\mathbb{X} = \{x_1, x_2, x_3, x_4\}$ be a set of variables, and suppose that the properties of interest are the (relational) sign of the product x_1x_2 and the parity of x_3 (both defined in Section 2). We slightly abuse notation by denoting as ρ_{SIGN} also its extension to pairs (v, t) where the sign of their product matters: e.g., $\rho_{\text{SIGN}}(3, -5) = ([\mathbf{neg}])$. In our formal framework, $\mathcal{A}$ is defined as $\langle\{x_1, x_2\} : \rho_{\text{SIGN}}, x_3 : \rho_{\text{PAR}}\rangle$. Let $\mu(x_1) = 1$, $\mu(x_2) = 2$, $\mu(x_3) = 3$, and $\mu(x_4) = 4$; then, $\mu \restriction^{\alpha}_{\mathcal{A}} \mathcal{X}$ comes to be $\mathcal{A} \circ \mu(\mathcal{X}) = \langle[\mathbf{pos}], [\mathbf{odd}]\rangle$.

The *abstract projection* operator modifies a state trajectory by removing all those states which do not contain occurrences or points of interest. If there is a state that contains an occurrence of interest, then its memory state is restricted via $\restriction^{\alpha}$ to the variables of interest, and only a property is considered for each tuple. In the following, the *abstract projection* $Proj^{\alpha}$ is formally defined.

Definition 4.6 (Abstract Projection). *Let $C_{\mathcal{A}} = (\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi, \mathcal{A})$, and $\mathcal{L} \subseteq \mathbb{L}$ such that $\mathcal{L} \neq \emptyset$ if $\psi = true$, $\mathcal{L} = \emptyset$ if $\psi = false$. For any $n \in \mathbb{L}$, $k \in \mathbb{N}$, $\mu \in \mathbb{M}$, we define a function $Proj^{0\alpha}$ as:*

$$Proj^{0\alpha}_{(\mathcal{X},\mathcal{O},\mathcal{L},\mathcal{A})}(n^k, \mu) \stackrel{def}{=} \begin{cases} \langle n^k, \mu \restriction^{\alpha}_{\mathcal{A}} \mathcal{X}\rangle & \text{if } \exists\langle n, K\rangle \in \mathcal{O}.\, k \in K \\ \langle n^k, \bot\rangle & \text{if } \nexists(n, K) \in \mathcal{O}.\, k \in K \text{ and } n \in \mathcal{L} \\ \varepsilon & \text{otherwise} \end{cases}$$

The abstract projection $Proj^{\alpha}$ is the extension of $Proj^{0\alpha}$ to sequences:

$$Proj^{\alpha}_{(\mathcal{X},\mathcal{O},\mathcal{L},\mathcal{A})}(\langle(n_1^{k_1}, \mu_1) \ldots (n_l^{k_l}, \mu_l)\rangle) = Proj^{0\alpha}_{(\mathcal{X},\mathcal{O},\mathcal{L},\mathcal{A})}(n_1^{k_1}, \mu_1) \circ \ldots \circ Proj^{0\alpha}{}_{(\mathcal{X},\mathcal{O},\mathcal{L},\mathcal{A})}(n_l^{k_l}, \mu_l)$$

$Proj^{0\alpha}$ takes a state from a state trajectory, and returns either one pair or an empty sequence ε. Abstract projection allows us to define all the semantic equivalence relations we need for representing the abstract forms of slicing.

Example 4.7. Consider the program P in Figure 7. Consider $\mathcal{I} = \mathbb{M}$ (meaning that we are considering static slicing), $\mathcal{X} = \{\mathsf{i}, \mathsf{s}\}$, $\mathcal{O} = \langle 8, \mathbb{N}\rangle$ (meaning that we check the value of variables of interest at each iteration of program point 8). Moreover, we consider $\mathcal{A} = \langle \mathsf{i} : \rho_{\text{SIGN}}, \mathsf{s} : \rho_{\text{PAR}}\rangle$. Then in Figure 8 we have the corresponding abstract projection (the concrete trace is given in the Appendix in Example 24). In this figure, we depict states as tuples of boxes: the first box contains the number of the executed program point (with the iteration counter as apex), while the other boxes indicate the values of each variable. A cross on a box means that the projection does not consider that variable or state. For instance, in this example we only care about states 6^i and 8^i; moreover, in states 6^i we are not interested in the values of variables, while in states 8^i we are interested in the sign of

```
1 read(n);                    1 read(n);
2 i := 1;                     2 i := 1;
3 s := 0;                     3 s := 0;
4 p := 1;                     4
5 while (i <= n) {            5 while (i <= n) {
6   s := s+i;                 6   s := s+i;
7   p := p*i;                 7
8   i := i+1; }               8   i := i+1; }
9 write(i,n,s,p);             9 write(i,n,s);
```

Program P | Program Q

Fig. 7. Program P and its slice.

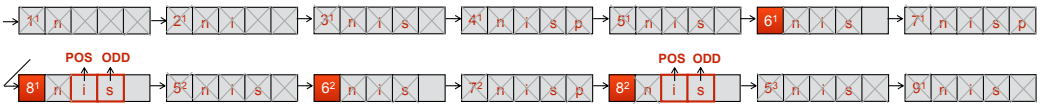


Fig. 8. Abstract trajectory projection for program P in Example 4.7

```
1 read(n);                    1 read(n);
2 read(s);                    2 read(s);
3 i := 1;                     3
4 while (i<=n) {              4
5   s := s+2*i;               5
6   i := i+1;   }             6
7 write(i,n,s);               7 write(n,s);
```

Fig. 9. Programs P and Q

i and in the parity of s (if we would be interested in the value of these variables we would have the value instead of their property, as it happens in the examples in the Appendix).

4.3. Abstract Unified Equivalence

The only missing step for completing the formal definition of abstract slicing in the formal framework is to characterize the functions mapping abstract slicing criteria to abstract semantic equivalence relations.

Definition 4.8 (Abstract Unifying Equivalence). *Let P and Q be executable programs, and $C_{\mathcal{A}} = (\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi, \mathcal{A})$ be an abstract criterion. Then P is* abstract-equivalent *to Q if and only if, for every $\mu \in \mathcal{I}$, it holds that $Proj^{\alpha}_{(\mathcal{X},\mathcal{O},\mathcal{L},\mathcal{A})}(\tau^{\mu}_{P}) = Proj^{\alpha}_{(\mathcal{X},\mathcal{O},\mathcal{L},\mathcal{A})}(\tau^{\mu}_{Q})$, where $\mathcal{L} = \mathbb{L}_P \cap \mathbb{L}_Q$ if $\psi =$ true. The function $\mathcal{E}_{\mathcal{A}}$ maps each criterion $C_{\mathcal{A}}$ to a corresponding abstract semantic equivalence relation.*

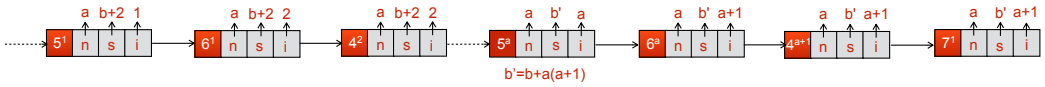
Therefore, a generic form of slicing can be represented as $(\sqsubseteq, \mathcal{E}_{\mathcal{A}})$. This can be used to formally define both traditional and abstract forms of slicing in the presented abstract formal framework, so that the latter comes to be a generalization of the original formal framework. The following examples show how it is possible to use these definitions in order to check whether a program is an abstract slice of another one.

Example 4.9. Consider programs P and Q in Figure 9. Let $C_{\mathcal{A}} = (\mathbb{M}, \{\mathtt{s}\}, \{\langle 7, \mathbb{N}\rangle\}, \mathit{false}, \langle \mathtt{s} : \rho_{\mathrm{PAR}}\rangle)$, meaning that we are interested in the parity of $\mathtt{s}$ ($\mathcal{A} = \langle \mathtt{s} : \rho_{\mathrm{PAR}}\rangle$) at the end of execution ($\mathcal{O} = \{\langle 7, \mathbb{N}\rangle\}$) for all possible inputs ($\mathcal{I} = \mathbb{M}$), in non-KL form. Since $Q \sqsubseteq P$, in order to show that Q is an abstract static slice of P with respect to $C_{\mathcal{A}}$, we have to show that $\langle P, Q\rangle \in \mathcal{E}_{\mathcal{A}}(C_{\mathcal{A}})$ holds. Let $\mu = \{n \leftarrow a, s \leftarrow b\}$

```
1 read(n);                    1 read(n);
2 s := 0;                     2 s := 0;
3 i := 1;                     3
4 while (i<=n) {              4
5   s := s+i;                 5
6   i := i+1; }               6
7 write(i,n,s);               7 write(n,s);
```

Fig. 10. Programs R and S

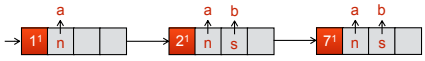
for some $a, b \in \mathbb{N}$ be an initial memory. Then, the trajectory of P from μ contains the following computation steps:



Applying $Proj^{\alpha}$ (with $\mathcal{L} = \emptyset$ since $\psi = \mathit{false}$) to τ^{μ}_{P} returns only the abstract value of the variable s at point 7 (due to $C_{\mathcal{A}}$):

$$\begin{aligned} Proj^{\alpha}_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau^{\mu}_{P}) &= \langle 7^1, \{n \leftarrow a, s \leftarrow b{+}a(a{+}1), i \leftarrow a{+}1\} \restriction^{\alpha}_{\mathcal{A}} \{\langle s \rangle\}) \rangle \\ &= \langle 7^1, \rho_{\mathrm{PAR}}(b{+}a(a{+}1)) \rangle = \langle 7^1, \rho_{\mathrm{PAR}}(b) \rangle \end{aligned}$$

Since we have $7^1 = \langle 7, 1 \rangle \in \mathcal{O}$, $Proj^{0\alpha}_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(7^1, \{n \leftarrow a, s \leftarrow b{+}a(a{+}1), i \leftarrow a{+}1\})$ returns $\langle 7^1, \mu \restriction^{\alpha}_{\mathcal{A}} \mathcal{X} \rangle = \langle 7^1, \{n \leftarrow a, s \leftarrow b{+}a(a{+}1), i \leftarrow a{+}1\} \restriction^{\alpha}_{\mathcal{A}} \{\langle s \rangle\} \rangle$. The abstract memory restricts the domain of μ to variables of interest, so we consider only the part of μ regarding s, i.e., $b{+}a(a{+}1)$. Hence, we have $\langle 7^1, \rho_{\mathrm{PAR}}(\langle b{+}a(a{+}1) \rangle) \rangle$. Since the parity of $b{+}a(a{+}1)$ only depends on the parity of b, being either a or $a{+}1$ even, the final result is $\langle 7^1, \rho_{\mathrm{PAR}}(b) \rangle$. Consider now the execution of Q from μ, which corresponds to the following state trajectory:

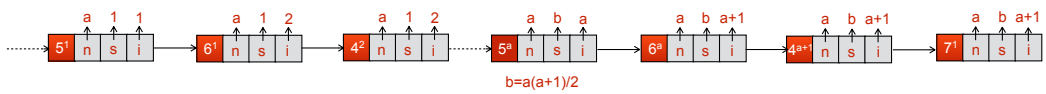


Applying $Proj^{\alpha}$ to τ^{μ}_{Q} gives:

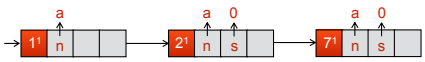
$$Proj^{\alpha}_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau^{\mu}_{Q}) = \langle 7^1, \{n \leftarrow a, s \leftarrow b\} \restriction^{\alpha}_{\mathcal{A}} \{\langle s \rangle\} \rangle = \langle 7^1, \rho_{\mathrm{PAR}}(b) \rangle$$

Therefore, $Proj^{\alpha}_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau^{\mu}_{P})$ is equal to $Proj^{\alpha}_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau^{\mu}_{Q})$. As μ is an arbitrary input, this equation holds for each $\mu \in \mathbb{M}$, so that $\langle P, Q \rangle \in \mathcal{E}_{\mathcal{A}}(C_{\mathcal{A}})$, and this implies that Q is an abstract static slice of P w.r.t. $C_{\mathcal{A}}$.

Example 4.10. Consider the programs R and S in Figure 10, and let $C_{\mathcal{A}}$ be $(\mathcal{I}, \{\mathsf{s}\}, \{\langle 7, \mathbb{N} \rangle\}, \mathit{false}, \langle \mathsf{s} : \rho_{\mathrm{PAR}} \rangle)$, where $\mathcal{I} = \{\ \mu \mid \mu(\mathsf{n}) \in 4\mathbb{Z}\ \}$; i.e., we are interested in the parity of s at the end of the execution for all inputs where n is a multiple of 4. Since $S \sqsubseteq R$, in order to show that S is an abstract conditioned slice of R w.r.t. $C_{\mathcal{A}}$, we have to show that $\langle R, S \rangle \in \mathcal{E}_{\mathcal{A}}(C_{\mathcal{A}})$ holds, namely that they have the same abstract projection. Let $\mu \in \mathcal{I}$ be an initial memory, and suppose $\mu(\mathsf{n}) = a = 4m$. The trajectory τ^{μ}_{R} of R from μ contains the following steps:



While executing S from μ gives the state trajectory τ_S^μ



Applying $Proj^\alpha$ to both state trajectories we have:

$$\begin{aligned} Proj^\alpha_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau_R^\mu) &= \langle 7^1, \{n \leftarrow a, s \leftarrow a(a+1)/2, i \leftarrow a+1\} \restriction^\alpha_{\mathcal{A}} \langle\{s\}\rangle\rangle = \langle 7^1, \rho_{\text{PAR}}(a(a+1)/2)\rangle \\ &= \langle 7^1, \rho_{\text{PAR}}(2m(4m+1))\rangle = \langle 7^1, 2\mathbb{Z}\rangle \\ &= \langle 7^1, \rho_{\text{PAR}}(0)\rangle = \langle 7^1, \{n \leftarrow a, s \leftarrow 0\} \restriction^\alpha_{\mathcal{A}} \langle\{s\}\rangle\rangle = Proj^\alpha_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau_S^\mu) \end{aligned}$$

Therefore, we have $Proj^\alpha_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau_R^\mu) = Proj^\alpha_{(\mathcal{X},\mathcal{O},\emptyset,\mathcal{A})}(\tau_S^\mu)$. Since μ is an arbitrary input from $\mathcal{I}$, this equation holds for each $\mu \in \mathcal{I}$, so that $\langle R, S\rangle \in \mathcal{E}_{\mathcal{A}}(C_{\mathcal{A}})$, and this implies that S is an abstract conditioned slice of R with respect to $C_{\mathcal{A}}$. It is worth noting that S is not a static abstract slice of R since, for all the input values $a \notin 4\mathbb{Z}$ for n, the parity of the final value of s is not necessarily even.

4.4. Comparing forms of Abstract Slicing

This section provides a formal theory allowing us to compare abstract forms of slicing between themselves, and with non-abstract ones. First of all, we show under which conditions an abstract semantic equivalence relation *subsumes* another one; analogously, we show when the form of (*abstract*) slicing, corresponding to the former equivalence relation, subsumes the form of (*abstract*) slicing corresponding to the latter one. Such results are necessary in order to obtain a precise characterization of the extension of the *weaker than* relation (whose original definition is recalled in the Appendix) to the abstract forms of slicing.

The following lemma shows under which conditions on the slicing criteria there is a relation of subsumption between two semantic equivalence relations. In the following, we denote $\overline{\sqsubseteq}$ the relation "more concrete than" in the lattice of abstract interpretations between tuples of abstractions. Formally, let us consider $\mathcal{A}^1 = \langle X_1^1 : \rho_1^1, \ldots, X_{k^1}^1 : \rho_{k^1}^1\rangle$ defined on the variables $\mathcal{X}^1$ and $\mathcal{A}^2 = \langle X_1^2 : \rho_1^2, \ldots, X_{k^2}^2 : \rho_{k^2}^2\rangle$ defined on the variables $\mathcal{X}^2$, such that $\mathcal{X}^1 \subseteq \mathcal{X}^2$, $k^1 \leq k^2$ and $\forall i \leq k^1$ we have $X_i^1 = X_i^2$, namely the variables in common are partitioned in the same way. Then $\mathcal{A}^2 \overline{\sqsubseteq} \mathcal{A}^1$ iff $\forall X_i \in \mathcal{X}^1.\ \rho_i^2 \sqsubseteq \rho_i^1$. Note that, for all the variables in $\mathcal{X}^2 \setminus \mathcal{X}^1$, the abstraction $\mathcal{A}^1$ does not require any particular observation, hence on these variables surely $\mathcal{A}^2$ is more precise. The following relation is such that, when both $\mathcal{A}^1$ and $\mathcal{A}^2$ are the identity on all the variables of interest, then the resulting criterion relation is the same proposed in [Binkley et al. 2006b] (see the Appendix for details) for characterizing the original formal framework.

LEMMA 4.11. *Let two abstract slicing criteria $C^1_{\mathcal{A}} = (\mathcal{I}^1, \mathcal{X}^1, \mathcal{O}^1, \psi^1, \mathcal{A}^1)$ and $C^2_{\mathcal{A}} = (\mathcal{I}^2, \mathcal{X}^2, \mathcal{O}^2, \psi^2, \mathcal{A}^2)$ be given. If (1) $\mathcal{I}^1 \subseteq \mathcal{I}^2$; (2) $\mathcal{O}^1 \subseteq \mathcal{O}^2$; (3) $\mathcal{X}^1 \subseteq \mathcal{X}^2$; (4) $\psi^1 \Rightarrow \psi^2$; and (5) $\mathcal{A}^2 \overline{\sqsubseteq} \mathcal{A}^1$ (denoted $C^1_{\mathcal{A}} \rightarrow_{\mathcal{A}} C^2_{\mathcal{A}}$), then $(\sqsubseteq, \mathcal{E}(C^1_{\mathcal{A}}))$ subsumes $(\sqsubseteq, \mathcal{E}(C^2_{\mathcal{A}}))$, i.e., for every P and Q such that $Q \sqsubseteq P$, $\langle P, Q\rangle \in \mathcal{E}(C^2_{\mathcal{A}})$ implies $\langle P, Q\rangle \in \mathcal{E}(C^1_{\mathcal{A}})$.*

PROOF. First of all, note that, if $\mathcal{A}^1 = \mathcal{A}^2 = \rho_{id}$, namely if we are considering concrete criteria, then $\rightarrow_{\mathcal{A}}$ collapses to the concrete relation defined in [Binkley et al. 2006a] (which is the $\rightarrow$ defined in Equation 2 in the Appendix). Hence, in this case, the results holds by [Binkley et al. 2006a].
Suppose $\langle P, Q\rangle \in \mathcal{E}(C^2_{\mathcal{A}})$ with $Q \sqsubseteq P$, namely Q slice of P w.r.t. $\mathcal{E}(C^2_{\mathcal{A}})$. This means that, for each $\mu_0 \in \mathcal{I}^2$ $Proj^\alpha_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(\tau_P^{\mu_0}) = Proj^\alpha_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(\tau_Q^{\mu_0})$, where $\mathcal{L}^2$ is defined as in Definition 4.8. This means that, for each state $\langle n^k, \mu\rangle$ in the trajectory $\tau_P^{\mu_0}$, whose

projection $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu)_P$[4] is not empty, there exists a state $\langle n^k,\mu'\rangle$ in $\tau^{\mu_0}_Q$ with the same projection. Let us consider now, $\mu \in \mathcal{I}^1 \subseteq \mathcal{I}^2$. We prove that on these states $Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu) = Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu)$ (and in this case there is a corresponding state in the trajectory of Q), or it is empty (and in this case also the state in Q has empty projection). Recall that

$$Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu) \stackrel{\text{def}}{=} \begin{cases} \langle n^k, \mu \restriction^{\alpha}_{\mathcal{A}^1} \mathcal{X}^1\rangle & \text{if } \exists\langle n,K\rangle \in \mathcal{O}^1.\ k \in K \\ \langle n^k,\bot\rangle & \text{if } \nexists(n,K) \in \mathcal{O}^1.\ k \in K \text{ and } n \in \mathcal{L}^1 \\ \varepsilon & \text{otherwise} \end{cases}$$

where also $\mathcal{L}^1$ is defined as in Definition 4.8. Note that, since we are considering both the criteria on the same pair of programs, we have also that $\psi^1 \Rightarrow \psi^2$ corresponds to saying that $\mathcal{L}^1 \subseteq \mathcal{L}^2$. At this point

- If $Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu)_P = \langle n^k, \mu \restriction^{\alpha}_{\mathcal{A}^1} \mathcal{X}^1\rangle$ then $\exists\langle n,K\rangle \in \mathcal{O}^1.\ k \in K$, but $\mathcal{O}^1 \subseteq \mathcal{O}^2$, hence $\langle n,K\rangle \in \mathcal{O}^2.\ k \in K$. This mean that $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu)_P = \langle n^k, \mu \restriction^{\alpha}_{\mathcal{A}^2} \mathcal{X}^2\rangle$, which by hypothesis is equal to $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu')_Q$, for a memory μ'. By definition and hypothesis, $\mu \restriction^{\alpha}_{\mathcal{A}^2} \mathcal{X}^2 = \mathcal{A}^2 \circ \mu(\mathcal{X}) = \mathcal{A}^2 \circ \mu'(\mathcal{X})$. Namely, $\langle \rho^2_1(\mu(X_1)),\ldots,\rho^2_{k^2}(\mu(X_{k^2}))\rangle = \langle \rho^2_1(\mu'(X_1)),\ldots,\rho^2_{k^2}(\mu'(X_{k^2}))\rangle$. Therefore, in particular, $\forall i \in [1,k^1] \subseteq [1,k^2]$ we have $\rho^2_i(\mu(X_i)) = \rho^2_i(\mu'(X_i))$, but by hypothesis $\rho^2_i \sqsubseteq \rho^1_i$, hence we also have $\rho^1_i(\mu(X_i)) = \rho^1_i(\mu'(X_i))$ (by properties of ucos). But then $\langle \rho^1_1(\mu(X_1)),\ldots,\rho^1_{k^1}(\mu(X_{k^1}))\rangle = \langle \rho^1_1(\mu'(X_1)),\ldots,\rho^1_{k^1}(\mu'(X_{k^1}))\rangle$, namely $\mathcal{A}^1 \circ \mu(\mathcal{X}) = \mathcal{A}^1 \circ \mu'(\mathcal{X})$. Hence

$$\begin{aligned} Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu)_P &= \langle n^k, \mu \restriction^{\alpha}_{\mathcal{A}^1} \mathcal{X}^1\rangle \\ &= \langle n^k, \mathcal{A}^1 \circ \mu(\mathcal{X})\rangle = \langle n^k, \mathcal{A}^1 \circ \mu'(\mathcal{X})\rangle \\ &= \langle n^k, \mu' \restriction^{\alpha}_{\mathcal{A}^1} \mathcal{X}^1\rangle = Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu')_Q \end{aligned}$$

- If $Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu)_P = \langle n^k,\bot\rangle$ then $\nexists(n,K) \in \mathcal{O}^1.\ k \in K$ and $n \in \mathcal{L}^1$. If $\nexists(n,K) \in \mathcal{O}^2.\ k \in K$ then $n \in \mathcal{L}^1 \subseteq \mathcal{L}^2$, then also $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu)_P = \langle n^k,\bot\rangle$ but then by hypothesis we have that there exists a memory μ' such that $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu')_Q = \langle n^k,\bot\rangle$. But then we also have $Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu)_Q = \langle n^k,\bot\rangle$.
  If $\exists(n,K) \in \mathcal{O}^2.\, k \in K$, then $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu)_P = \langle n^k, \mu \restriction^{\alpha}_{\mathcal{A}^2} \mathcal{X}^2\rangle$, but then there exists μ' such that also in Q we have $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu')_Q = \langle n^k, \mu' \restriction^{\alpha}_{\mathcal{A}^2} \mathcal{X}^2\rangle$. But then, the same memory, in $C^1_{\mathcal{A}}$ keep the program point but loses the state observation because $\nexists(n,K) \in \mathcal{O}^1.\ k \in K$ and $n \in \mathcal{L}^1$, hence $Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu')_Q = \langle n^k,\bot\rangle$.
- Finally, if $Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu)_P = \varepsilon$, then $\nexists(n,K) \in \mathcal{O}^1.k \in K$ and $n \notin \mathcal{L}^1$. But this means that, even if there exists μ' such that we have the state $\langle n^k,\mu'\rangle$ in the trajectory of Q, also in this case we have $Proj^{0\alpha}_{(\mathcal{X}^1,\mathcal{O}^1,\mathcal{L}^1,\mathcal{A}^1)}(n^k,\mu')_Q = \varepsilon$.

□

This lemma shows how it is possible to find the relationship (in the sense of subsumption) between two semantic equivalence relations determined by two abstract slicing criteria. In the following, abstract notions of slicing will be denoted by adding an $\mathcal{A}$; e.g., $\mathcal{AS}$ denotes static abstract slicing, whereas $\mathcal{AD}$ denotes dynamic abstract slicing. By using this

[4]The notation $Proj^{0\alpha}_{(\mathcal{X}^2,\mathcal{O}^2,\mathcal{L}^2,\mathcal{A}^2)}(n^k,\mu)_P$ means that we are projecting a state of the computation of P.

lemma we can show that, given a slicing criterion $C_{\mathcal{A}}$, all the abstract equivalence relations introduced in Sec. 4.3 subsume the corresponding non-abstract equivalence relations $\mathcal{S}(C_{\mathcal{A}})$, $\mathcal{D}(C_{\mathcal{A}})$ and $\mathcal{C}(C_{\mathcal{A}})$. Furthermore, by using this lemma we can show that $\mathcal{AD}(\mathcal{C}_{\mathcal{A}})$ subsumes $\mathcal{AC}(\mathcal{C}_{\mathcal{A}})$, which in turns subsumes $\mathcal{AS}(\mathcal{C}_{\mathcal{A}})$.

THEOREM 4.12. *[Binkley et al. 2006b] Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be semantic equivalence relations such that $\mathcal{R}_2$ subsumes $\mathcal{R}_1$. Then, for every P and Q, we have $\langle P, Q\rangle \in (\sqsubseteq, \mathcal{R}_1) \Rightarrow \langle P, Q\rangle \in (\sqsubseteq, \mathcal{R}_2)$.*

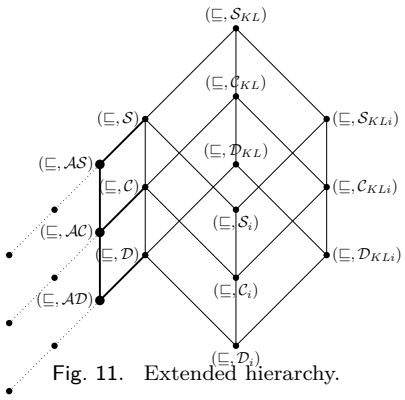


Fig. 11. Extended hierarchy.

Fig. 11 shows the non-SIM hierarchy obtained by enriching the hierarchy in Fig. 25 with standard forms of *abstract static slicing*, *abstract dynamic slicing*, and *abstract conditioned slicing*. In general, we can enrich this hierarchy with any abstract form of slicing simply by using the comparison notions defined above. Non-abstract forms are particular cases of abstract forms of slicing, as they can be instantiated by choosing the identity property, ρ_{ID}, for each variable of interest. Hence, non-abstract forms are the "strongest" forms, since, for each property ρ, we have $\rho_{\text{ID}} \sqsubseteq \rho$. Moreover, if parameters $M, \mathcal{X}, \mathcal{O}, \psi$ are fixed, and $\mathcal{A}$ is made less precise or more abstract (i.e., the information represented by the property is reduced), then the abstract slicing form becomes weaker, as suggested by dotted lines in Figure 11.

5. PROGRAM SLICING AND DEPENDENCIES

The previous sections introduced a formal framework containing different notions of program slicing. In particular, we observed that a kind of slicing is a pair consisting of a syntactic preorder and a semantic equivalence relation [Binkley et al. 2006a]. After discussing how the notion of "to be a slice of" can be formally defined, the focus will shift to how to *compute* a slice given a program and a slicing criterion. Again, among all the possible definitions of slicing, we are interested in slices obtained by erasing statements from the original program, i.e., the slice is related to the original program by the syntactic ordering relation $\sqsubseteq$. Given a slicing criterion, the idea is to keep all the statements affecting the *semantic equivalence relation* defined by the criterion. In other words, we should have to *translate* the formal definition into a characterization of which statements has to be kept in a slice, or vice versa which statements can be erased, in order to preserve the semantic equivalence defining the chosen notion of slicing. Intuitively, we have to keep all the statements *affecting* the semantics defined by the chosen slicing criterion.

The standard approach for characterizing slices and the corresponding relation *being slice of* is based on the notion of Program Dependency Graph [Horwitz et al. 1989; Reps and Yang 1989], as described by Binkley and Gallagher [Binkley and Gallagher 1996]. *Program*

Dependency Graphs (PDGs) can be built out of programs, and describe how data propagate at runtime. In program slicing, 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 5.1. Consider the program in Figure 12 and the derived PDG (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.

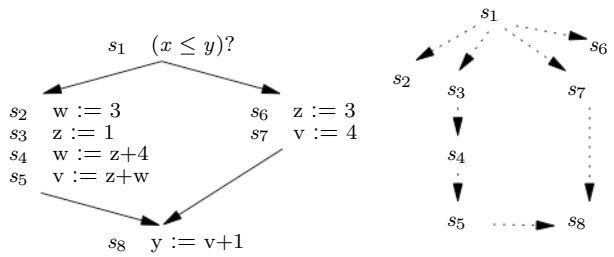


Fig. 12. PDG example.

Formally, a *Program Dependence Graph* [Gallagher and Lyle 1991] $\mathcal{G}_P$ for a program P is a directed graph with nodes denoting program components and edges denoting *dependencies* between components. The nodes of $\mathcal{G}_P$ represent the assignment statements and control predicates in P. In addition, nodes include a distinguished node called *Entry*, denoting where the execution starts. An edge represents either a *control dependency* or a *flow* (*data*) *dependency*. Control dependency edges $u \longrightarrow_c v$ are such that (1) u is the entry node and v represents a component of P that is not nested within any control predicate; or (2) u represents a control predicate and v represents a component of P immediately nested within the control predicate represented by u. Flow dependency edges $u \longrightarrow_f v$ are such that (1) u is a node that defines the variable x (usually an assignment), (2) v is a node that *uses* x, and (3) control can reach v from u via an execution path along which there is no intervening re-definition of x.

Unfortunately, there is clearly a gap between the definition of slicing given in Definition 3.1 and the standard implementation based on program dependency graphs (PDG) [Horwitz et al. 1989; Reps 1991]. This happens because slicing and dependencies are usually defined at *different levels* of approximation. In particular, we can note that the slicing definition in the formal framework defines slicing by requiring the same *behavior*, with respect to a criterion, between the program and the slice, i.e., we are specifying what is *relevant* as a *semantic* requirement. On the other hand, dependency-based approaches 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 defined at the *syntactic* level. The idea presented in this paper consists, first of all, in 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. This way, we can precisely identify the semantic definition of slicing corresponding to a given dependency-based algorithm, characterizing so far the loss of precision of a given algorithm w.r.t. the semantic definition.

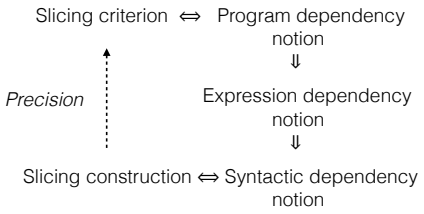


Fig. 13. Schema of dependency-based slicing notions.

In Figure 13 we show these relations. In particular, starting from the criterion, we can define an equivalent notion of dependency which allows us to identify which variables should be kept in a slice, affecting the whole program semantics (program dependency notion).

DEFINITION 5.2 ((SEMANTIC) PROGRAM DEPENDENCY). *Let $C = (\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi)$ be a slicing criterion, and P be a program. The program depends on x, denoted $x \rightsquigarrow_C P$ iff*

$$\exists \mu_1, \mu_2 \in \mathcal{I}.\, \forall y \neq x.\, \mu_1(y) = \mu_2(y) \ \wedge \ \mathit{Proj}_{(\mathcal{X},\mathcal{O},\mathcal{L})}(\tau_P^{\mu_1}) \neq \mathit{Proj}_{(\mathcal{X},\mathcal{O},\mathcal{L})}(\tau_P^{\mu_2})$$

This means that the variable x affects the observable semantics of P.

Unfortunately, this characterization is not effective due to undecidability of the program semantics. In particular, the amount of traces to compare could be infinite, and also the traces themselves could be infinite. Hence, we consider a stronger notion of dependency that looks for *local* semantic dependencies, identifying all the variables affecting at least one expression used in the program (expression dependency notion), and this is precisely the *semantic* generalization of the syntactic dependency notion used, for instance, in PDG-based algorithm for slicing. In other words we characterize when a variable affects the semantics of an expression in P.

Our idea is to make semantic the standard notion of syntactic dependency, by substituting the notion of *uses* with the notion of *depends on* [Giacobazzi et al. 2012]. In order to obtain this characterization, we have to find which variables might affect the evaluation of the expression e in the assignment $\mathsf{z}{:=}\mathsf{e}$ or in a control statement guarded by e, i.e., which variables belong to the set $\text{REL}(e)$ of the variables *relevant* to the evaluation of e. As already pointed out, standard syntactic dependency calculi compute $\text{REL}(e)$ as $\text{VARS}(e)$.

DEFINITION 5.3 ((SEMANTIC) EXPRESSION DEPENDENCY). *Let $C = (\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi)$ be a slicing criterion. Let $x \in \mathbb{X}$, $\mathcal{Y} \subseteq \mathbb{X}$.*

$$\begin{array}{l} x \rightsquigarrow_C e \ \Leftrightarrow \ \exists \sigma_1, \sigma_2 \in \Sigma.\, \forall y \neq x.\, \sigma_1(y) = \sigma_2(y) \wedge \ [\![e]\!](\sigma_1) \neq [\![e]\!](\sigma_2) \\ \mathcal{Y} \rightsquigarrow_C e \ \Leftrightarrow \ \exists y \in \mathcal{Y}.\ y \rightsquigarrow_C e \end{array}$$

The formulation of $x \rightsquigarrow_C e$ can be rewritten as

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

PROPOSITION 5.4. *Let C be a slicing criterion. If $x \rightsquigarrow_C P$ then there exists e in P such that $x \rightsquigarrow_C e$.*

PROOF. By contradiction. If for each e in P we have $x \not\rightsquigarrow_{C_\mathcal{A}} e$, then no expression in P depends on x. Therefore, independently from x, P provides precisely the same results. □

By using this notion of dependency, we can characterize the subset $\text{REL}(e) \subseteq \text{VARS}(e)$ containing exactly those variables which are *semantically* relevant for the evaluation of e. This way, we obtain a notion of dependency which allows us to derive more precise slices, i.e., to remove statements that a merely syntactic analysis would leave.

Example 5.5. Consider the program P:

```
1 x:=e_x ;
2 y:=e_y ;
3 w:=e_w ;
4 z:=w+y+2(x^2)−w;
```

where e_x, e_y and e_w are expressions. We want to compute the static slice Q of P affecting the final value of z (i.e., the *slicing criterion* $C = (\mathbb{M}, \{\mathsf{z}\}, \langle 4, \mathbb{N}\rangle, \mathit{false})$ is interested in the final value of z). If we consider the traditional notion of slicing, then it is clear that we can erase line 3 without changing the final result for z. However, in the usual syntactic 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 $\mathsf{w}+\mathsf{y}+2(\mathsf{x}^2)-\mathsf{w}$ does not depend on the possible variations of w, which implies that we are able to erase line 3 from the slice.

Next we show how the PDG-based approach to slicing can be modified in order to cope with semantic slicing. The PDG approach is based on the computation of the set of variables *used* in a expression e. In the following, we explore if this set can be rewritten by considering a semantic form of dependency.

Hence, let us define the new notion of *semantic* PDG, where all the flow dependencies are *semantic*, i.e., we substitute the flow edges defined above with semantic flow dependency edges $u \longrightarrow_{sf} v$ which are such that (1) u is a node that defines the variable x (usually an assignment), (2) v is a node containing an expression e such that $x \rightsquigarrow_C e$ (where C is the criterion with respect with we are computing the slice), and (3) control can reach v from u via an execution path along which there is no intervening re-definition of x. A (semantic) flow path is a sequence of (semantic) flow edges.

PROPOSITION 5.6. *Let P be a program and C be a slicing criterion. Let $\mathcal{G}_P$ the PDG with flow dependency edges $\longrightarrow_f$, and $\mathcal{G}^s_P$ be the semantic PDG where the flow dependency edge are semantic $\longrightarrow_{sf}$. If $u \longrightarrow_{sf} v$ then $u \longrightarrow_f v$.*

PROOF. Trivially, since if $x \rightsquigarrow_C e$, then e must use the variable x. □

In principle, a (backward) slice is composed by all the statements (i.e., nodes) such that there exists a path from the corresponding node to the relevant (according to the slicing criterion) use of a variable of interest (in the criterion) [Reps and Yang 1989]. In other words, we follow backward the (semantic) flow edges from the nodes identified by the criterion, and we keep all the nodes/statements we reach. Hence, the criterion defines the edges that we can follow in order to compute the slice.

By using the semantic flow dependency edges, we can draw a new *semantic* PDG containing less flow edges, i.e., only those corresponding to semantic dependencies. At this point, the type of slicing (either static, dynamic or conditional) characterized by the criterion decides which nodes can be kept in the PDG.

THEOREM 5.7. *Let $\mathcal{C} = \langle \mathcal{I}, \mathcal{X}, \mathcal{O}, \psi\rangle$ be a slicing criterion. Let P be a program and $\mathcal{G}^s_P$ its* semantic PDG*, i.e., a PDG whose flow edges are $\longrightarrow_{sf}$. Let Q the subprogram of P containing all the statements corresponding to nodes such that there exists a semantic flow path in $\mathcal{G}^s_P$ from them to a node in $\mathcal{N}_\mathcal{O} \stackrel{def}{=} \{\ n\ \mid \exists\langle n, K\rangle \in \mathcal{O}\ \}$. Then Q is a slice w.r.t. the criterion $\mathcal{C}$.*

PROOF. Note that, in the PDG construction all the flow edges ($\longrightarrow_{sf}$) satisfy a notion of semantic dependencies holding for all inputs by definition, while all the other edges are syntactic, hence its construction is independent from the input set $\mathcal{I}$, meaning that any slice

computed by using the semantic PDG holds for any possible input memory in $\mathbb{M}$. Moreover, since we simply collect the statements potentially affecting the program observation, we cannot decide which iteration to observe, for this reason each program point is taken in the slice independently from the iteration to observe, for this reason the obtained slice will provide the same result for any possible iteration, i.e., the set of interesting points to observe are $\mathcal{N}_\mathcal{O} \times \{\mathbb{N}\} = \{ \langle n, \mathbb{N}\rangle \mid n \in \mathcal{N}_\mathcal{O} \}$. Finally, by construction we cannot have statements executed in the slice which are not executed in the original programs. Hence, the criterion enforced by the PDG slice construction is $C^{\text{PDG}} = (\mathbb{M}, \mathcal{X}, \mathcal{N}_\mathcal{O} \times \{\mathbb{N}\}, \mathit{true})$ (direct consequence of the *Slicing theorem* in [Reps and Yang 1989]), which for each $\mathcal{I} \subseteq \mathbb{M}$ and $\{K_n\}_{n\in\mathcal{N}_\mathcal{O}} \subseteq \wp(\mathbb{N})$ is a slice also w.r.t. the criterion $(\mathcal{I}, \mathcal{X}, \{\langle n, K_n\rangle \mid n \in \mathcal{N}_\mathcal{O}\}, \psi)$, by [Binkley et al. 2006a] (Equation 2 in the Appendix). Hence, we have that the results is a slice w.r.t. C [Binkley et al. 2006a]. □

We have pointed out so far the difference between syntactic and semantic dependencies: it can be the case that a variable syntactically appears in an expression without affecting its value. Actually, one could argue that the case is not very likely to happen: the possibility to find an assignment like x:=y−y in code written by a professional software engineer is remote to the very least. However, when it comes to abstract dependencies, the picture is quite different, and we could even say that, in the present work, (concrete) semantic dependencies have been mainly introduced to prepare the discussion about their abstract counterpart. Indeed, it is much more likely that some variables are not semantically relevant to an expression if the value of interest is an abstract one, e.g., the parity or the sign of a numeric expression, or the nullity of a pointer. This justifies the definition of a semantic notion of dependency at the abstract level.

6. ABSTRACT DEPENDENCIES

This section discusses the problem of defining and computing abstract dependencies, which allow capturing the dependency relation between variables with respect to a given abstract criterion. The previous section formalized this relation in the concrete semantic case; the following example takes it to the abstract level.

Example 6.1. Consider the expression $e = 2\mathsf{x}^2+\mathsf{y}$: although both variables are semantically relevant to the result, only y can affect its parity, since $2\mathsf{x}^2$ will always be even. On the other hand, note that both variables are relevant to the sign of e, in spite of the positivity of x^2. In fact, given a negative value for y, a change in the value of x can alter the sign of the entire expression.

The notion of *semantic program dependency* can be easily extended to abstract criteria by changing the projection considered. In this case, we write $x \rightsquigarrow_{C_\mathcal{A}} P$, meaning that x has effect on the abstract projection of P determined by $C_\mathcal{A}$. Computing abstract dependencies is still undecidable; hence, again, we need an approximation of the semantic notion.

6.1. Abstract slicing and dependencies

Previously, we defined concrete semantic dependencies by identifying those variables that affect the final observation of an expression. Analogously, in order to define abstract semantic dependencies, we need to consider the *abstract* interference between a property of a variable and a property of an expression.

The definition below follows the same philosophy as *narrow abstract non-interference* [Giacobazzi and Mastroeni 2004a; Mastroeni 2013], where abstractions for observing input and output are considered, but these abstractions are observations of the *concrete* executions.

Definition 6.2 ((Abstract) Semantic expression dependency, *Ndep*). *Consider the ucos $\rho \in uco(\wp(\mathbb{V}))$ and $\overline{\eta} \in uco(\wp(\mathbb{V}))^n$, where n is the number of variables, i.e.,*

$\overline{\eta}$ is a tuple such that η_y is the property on the variable y[5]. Then:

$$x \overset{\overline{\eta}}{\rightsquigarrow} e \;\Leftrightarrow\; \exists \sigma_1, \sigma_2 \in \Sigma\,.\; \Big(\forall y \neq x\,.\, \eta_y(\sigma_1(y)) = \eta_y(\sigma_2(y)) \;\wedge\; \rho([\![e]\!](\sigma_1)) \neq \rho([\![e]\!](\sigma_2))\Big)$$

This notion is a generalization of Definition 5.3 where we abstract the observation of the result by applying ρ, and the information that we fix about all the variables different from x by using $\overline{\eta}$. Still, this notion characterizes whether the variation of the value of x affects the abstract evaluation in ρ of e.

As an important result, we have $x \overset{\overline{\eta}^1}{\rightsquigarrow} e \Leftrightarrow x \overset{\overline{\eta}^2}{\rightsquigarrow} e$ whenever ρ^1 and ρ^2 induce the same partitions (Section 2.3), either on values or tuples of values. This happens because, in Definition 6.2, both abstractions are only applied to singletons. In the following, only partitioning domains will be considered since it is straightforward to note that $x \overset{\overline{\eta}}{\rightsquigarrow} e$ is affected only by $\Pi(\rho)$, rather than by ρ itself.

When dealing with abstractions and abstract computations, some more issues have to be taken into account. Consider the program in Example 5.5, and consider the ρ_{PAR} property (Section 2.2) for all variables on both input and output. If we compute the set of variables on which the parity of e = w+y+2(x^2)−w depends, then we can observe that e is still independent from w, but is also independent from any possible variation of x. At a first sight, the parity of w+y+2(x^2)−w is independent from x just because 2(x^2) is constantly even. However, it is not only a matter of constancy: 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* (Section 2.2), i.e., precise, with respect to the abstract domain considered (ρ_{PAR}). In particular, when we deal with abstract domains which are complete for the considered operations, then it is enough to look at the abstract value of variables in order to compute dependencies. Indeed, consider the ρ_{SIGN} domain (Section 2.2): even if the sign of 2(x^2) is constantly positive, the final sign of z might be affected by a concrete variation of x (e.g., consider y = −4 and two executions in which x is, respectively, 1 and 5). Therefore, x has to be considered relevant, although the sign of 2(x^2) (the only sub-expression containing x) is constant. This can be also derived by considering the logic of independencies from [Amtoft and Banerjee 2007] since, by varying the value of x, we can change the sign of e.

Unfortunately, the notion of abstract dependencies given in Definition 6.2 is not suitable for weakening the PDG approach, as we have done in the concrete semantic case. Let us explain the problem in the following example.

Example 6.3. Consider the program P =C; x:=(y>0?0:1);, where C is some code fragment and the expression b?e1:e2 evaluates to e1 if b is true, and to e2 otherwise. Suppose the criterion requires the observation of the parity of x at this program point, i.e., $\mathcal{A} = \langle x : \rho_{\text{PAR}} \rangle$. Then, it is straightforward to observe that the expression depends on y, but we would like to be more specific (being in the context of abstract slicing), and we can observe that it is the sign of y that affects the parity of the expression, and therefore of x. Therefore, in the code C we should look for the variables affecting not simply y (as expected in standard slicing approaches), but more specifically the *sign* of y, a requirement which is not considered in the abstract criterion.

This example shows that, if we aim at computing abstract dependencies precisely, we would need an algorithm able to keep trace (backwards) not only of the different variables that become of interest (i.e., affecting the desired criterion), but also of the different properties to observe on such variables. This means that, while for concrete semantic program dependencies we can provide a definition depending only on the criterion, this is not possible in an abstract context, because each flow edge should be defined depending on abstract

[5]For simplicity, we do not consider relational abstract domains in this section.

properties potentially different from those in the abstract criterion, and which should be characterized dynamically backwards starting from the criterion. Unfortunately, this is not possible in Definition 6.2, where we always look for the variation of the value rather than an abstract property of x. Hence, using this notion for substituting the semantic dependency in PDGs would not give the desired results.

These observations make clear that, if we aim at constructively characterize abstract slicing by means of the abstract dependency notion provided in Definition 6.2, we need to build from scratch a systematic approach for characterizing abstract slicing. The first step in this direction is a computable approximation of the abstract dependencies of Definition 6.2.

6.2. A constructive approach to Abstract Dependencies

By means of the (uco-dependent) definition of operations on abstract values, it is possible to automatically obtain (an over-approximation of) the set of relevant variables. The starting point is the *brute-force* approach which uses the abstract version of concrete operations, and explicitly *goes into* the quantifiers involved in Definition 6.2.

Example 6.4. Consider the program in Example 5.5 and let $\eta_y = \rho_{\text{PAR}}$. In order to decide whether $\mathsf{x} \overset{\overline{y}}{\rightsquigarrow}_{\rho} e$ holds for some ρ, the brute-force approach considers the abstract evaluation of e in all contexts where all variables different from x does not change, up to $\overline{\eta}$, while x may change. In this example, this boils down to consider pairs of memories where y has the same parity (with no information about sign) whereas x changes freely, and see whether the final values of e agree on ρ. Suppose $\rho_{\text{PAR}}(\mathsf{y}) = [\mathbf{even}]$ (meaning that it is even but the sign is unknown) and $\rho = \rho_{\text{SIGN}}$; then, we should compute the abstract value of e for each possible value of x. We can easily find σ_1 and σ_2 such that

$$\rho_{\text{SIGN}}(2 * \sigma_1(\mathsf{x})^2 + \sigma_1(\mathsf{y})) = [\mathbf{neg}]\,, \qquad \rho_{\text{SIGN}}(2 * \sigma_2(\mathsf{x})^2 + \sigma_2(\mathsf{y})) = [\mathbf{pos}]$$

even if $\rho_{\text{PAR}}(\sigma_1(\mathsf{y})) = \rho_{\text{PAR}}(\sigma_2(\mathsf{y}))$ (for instance $\sigma_1(\mathsf{y}) = \sigma_2(\mathsf{y}) = -4$ while $\sigma_1(\mathsf{x}) = 1$ and $\sigma_2(\mathsf{x}) = 5$). It is clear that the sign of e may depend on the value of x since to "fix" the abstract property of the other variables is not enough to "fix" the final value of e with respect to ρ. On the other hand, the parity of e does not depend on x: if we fix the property ρ_{PAR} of y, for instance to $[\mathbf{even}]$ (but it holds also for the other abstract values), then

$$2 * [\mathbf{negodd}]^2 + [\mathbf{even}] = [\mathbf{even}]\,, \qquad 2 * [\mathbf{poseven}]^2 + [\mathbf{even}] = [\mathbf{even}] \qquad \ldots$$

That is to say, if we fix all the (abstract values of the) variables but x, then the parity of the result does not change, so that the variation of x does not affect the parity of e.

In the following, we introduce an algorithm to improve the computational complexity of the brute-force approach, especially on bigger ucos, and when (1) several variables are involved in expressions, and (2) a significant part of them is irrelevant.

6.2.1. Checking Ndep. We discuss how to constructively compute dependencies. Unfortunately, in static analysis, the concrete semantics cannot be used directly as it appears in Definition 6.2, so that we need to design an over-approximation. The following definition introduces a stronger notion of dependency based on a sound abstract semantics $[\![\cdot]\!]^{\rho}$ (Section 2.4), which approximates *Edep*.

DEFINITION 6.5 (*Atom-dep*). *An expression* e atom-depends *on* x *(written* $x \overset{\overline{y}}{\rightsquigarrow}_{\rho}{}_{\text{AT}} e$*) with respect to* $\rho \in uco(\wp(\mathbb{V}))$ *and* $\overline{\eta} \in uco(\wp(\mathbb{V}))^n$ *(*n *is the number of variables) if and only if there exist* $\sigma_1, \sigma_2 \in \Sigma$ *such that*

$$\forall y \neq x . \eta_y(\sigma_1(y)) = \eta_y(\sigma_2(y)) \quad \wedge \quad \neg \textsc{Atom}_{\rho}\left([\![e]\!]^{\rho}(\{\sigma_1, \sigma_2\})\right)$$

Since domains are partitioning, the *non-atomicity* requirement $\neg \textsc{Atom}_{\rho}(\cdot)$ amounts to say that all abstract evaluations of e with respect to ρ, starting from different values for x,

may not be abstracted to the same abstract value (this is the crucial issue in *Ndep*), i.e., σ_1 and σ_2 may lead to different abstract values for e. Next result shows that *Atom-dep* is an approximation of *Ndep*, since *Ndep* implies *Atom-dep*, meaning that *Atom-dep* may add false dependencies.

PROPOSITION 6.6. *Consider the abstractions $\rho \in uco(\wp(\mathbb{V}))$ and $\overline{\eta} \in uco(\wp(\mathbb{V}))^n$, where n is the number of variables, i.e., $\overline{\eta}$ is a tuple of properties. For every e and x, $x \rightsquigarrow_{\overline{\eta}} e$ implies $x \rightsquigarrow_{\overline{\eta}}{}_{\text{AT}} e$.*

PROOF. Suppose $x \rightsquigarrow_{\overline{\eta}} e$, i.e., there exist two states σ_1 and σ_2 such that (1) $\forall y \neq x.\ \eta_y(\sigma_1) =_y \eta_y(\sigma_2)$; and (2) $\rho([\![e]\!](\sigma_1)) \neq \rho([\![e]\!](\sigma_2))$. Then we have to prove $x \rightsquigarrow_{\overline{\eta}}{}_{\text{AT}} e$, i.e., there exist $\sigma_1, \sigma_2 \in \Sigma$ such that (3) $\forall y \neq x.\eta_y(\sigma_1) =_y \eta_y(\sigma_2)$; and (4) $\neg \text{ATOM}_\rho\left([\![e]\!]^\rho(\{\sigma_1, \sigma_2\})\right)$. Since conditions (1) and (3) are the same, we have to prove that (2) implies (4).

Consider $\sigma_1, \sigma_2 \in \Sigma$ satisfying (1) and such that $\rho([\![e]\!](\sigma_1)) \neq \rho([\![e]\!](\sigma_2))$; then, $\rho([\![e]\!](\sigma_1)), \rho([\![e]\!](\sigma_2)) \in \rho([\![e]\!](\{\sigma_1, \sigma_2\})) \subseteq \rho([\![e]\!](\rho(\{\sigma_1, \sigma_2\}))) = [\![e]\!]^\rho(\{\sigma_1, \sigma_2\})$. At this point, since $[\![e]\!]^\rho(\{\sigma_1, \sigma_2\})$ is greater than two different values of ρ, then, by definition, it cannot be an atom of ρ. □

A simple example where Atom-dep introduces false dependencies is the expression $e =$ x∗x + x∗x + 1 where $\rho = \rho_{\text{PARSIGN}}$ ($\overline{\eta}$ is irrelevant in this case). It is clear that the result of the expression is always positive and odd, so that $\rho([\![e]\!](\sigma)) = [\mathbf{poseven}]$ for every σ. Therefore, according to Definition 6.2, there is no abstract dependency of e on x with respect to ρ. On the other hand, it is possible to take two concrete states σ_1 and σ_2 where x has different values and, in addition, such values are also different as regards the sign and/or the parity. For example, suppose $\sigma_1(x) = 1$ and $\sigma_2(x) = -4$: the abstract state σ^ρ which is the result of abstracting the set $S = \{\sigma_1, \sigma_2\}$ is such that $\sigma^\rho(x) = [\mathbf{top}]$ since 1 and -4 have different parity and different sign. Computing the abstract semantics $[\![e]\!]^\rho(\sigma^\rho)$ boils down to evaluate e at the abstract level with [**top**] as the value for x. Since $[\mathbf{top}] * [\mathbf{top}] + [\mathbf{top}] * [\mathbf{top}] + [\mathbf{posodd}] = [\mathbf{top}] + [\mathbf{top}] + [\mathbf{posodd}] = [\mathbf{top}]$[6], then there is dependence of e on x according to Definition 6.5 since [**top**] is obviously not an atom.

Starting from this new approximated notion, the idea is to provide an algorithm which over-approximates the set of variables relevant for a given expression e. For simplicity, the abstraction observed on the output is supposed to be the same as the one on each input variable (i.e., each component of the tuple $\overline{\eta}$ is exactly the ρ observed on the output, so that $\overline{\eta}$ will be denoted as $\overline{\rho}$ or even left implicit in the following). The idea is to start from an empty set of non-relevant variables X, and incrementally increase this set adding all those variables that *surely* are not relevant. Finally, the complement of such set is returned, which is an over-approximation of relevant variables.

In order to check the dependency relation, we aim at checking whether a change in the values of a variable makes a difference in the evaluation of the expression. Dependencies are computed according to *Atom-dep*, in order to approximate *Ndep*. In a brute-force approach, *Atom-dep* would be verified by checking for each σ^ρ associating atomic values to variables we have that $\text{ATOM}_\rho\left([\![e]\!]^\rho(\sigma^\rho)\right)$ is always the same atom.

Example 6.7. Let e be an expression involving variables x, y and z, and $\rho = \rho_{\text{PARSIGN}}$. In principle, in order to compute the set of ρ-dependencies on e, we must compute $[\![e]\!]^\rho$ on every

[6] [**posodd**] is the abstraction of the constant 1, which is positive and odd.

possible atomic value[7] of x, y and z, i.e., $[\![e]\!]^\rho$ must be computed $5^3 = 125$ times. y is *not* relevant to e if, for any abstract values $\mathrm{v}_x, \mathrm{v}_z \in \textsc{Atoms}(\rho)$, there exists an atomic abstract value $\mathrm{u} \in \textsc{Atoms}(\rho)$ such that $\forall \mathrm{v} \in \textsc{Atoms}(\rho)\,.\ \mathrm{u} = [\![e]\!]^\rho(\{x \leftarrow \mathrm{v}_x, y \leftarrow \mathrm{v}, z \leftarrow \mathrm{v}_z\})$. This amounts to say that changing the value of y does not affect e, since we require the same atom to be computed for each possible abstract value of y and z. If the result is not atomic, it means that there may[8] exist two different abstract results for different values of y and z.

However, it is possible to be smarter:

- *Excluding states:* consider dependencies of e in Example 6.7, computed at program point n. Suppose $[\![\cdot]\!]^\rho$ (used as a tool to infer invariant properties, as discussed in Section 2.4) is able to infer, at point n, that the abstract state σ^ρ_n such that $\sigma^\rho_n(\mathrm{y}) = [\mathbf{posodd}]$ correctly approximates the value of variables at n. Then, we only need to consider states of the form $\{x \leftarrow \mathrm{v}_x, y \leftarrow [\mathbf{posodd}], z \leftarrow \mathrm{v}_z\}$ as inputs for $[\![e]\!]^\rho$ (now considered as the abstract computation of expressions, according to Definition 6.5) at n.
- *Computing on non-atomic states:* let $E = \{[\mathbf{poseven}], [\mathbf{zero}], [\mathbf{negeven}]\}$ and $O = \{[\mathbf{posodd}], [\mathbf{negodd}]\}$. In this case,

$$\forall \mathrm{v}' \in E, \mathrm{v}'' \in O.\ [\![e]\!]^\rho(\{x \leftarrow \mathrm{v}_x, y \leftarrow \mathrm{v}', z \leftarrow \mathrm{v}''\}) \leq \mathrm{u}$$

  is implied by the more general result

$$[\![e]\!]^\rho(\{x \leftarrow \mathrm{v}_x, y \leftarrow [\mathbf{even}], z \leftarrow [\mathbf{odd}]\}) \leq \mathrm{u}$$

  since E and O are partitions, respectively, of [**even**] and [**odd**], and $[\![e]\!]^\rho$ is monotone: $\sigma^\rho_1 \leq \sigma^\rho_2$ implies $\Rightarrow [\![e]\!]^\rho(\sigma^\rho_1) \leq [\![e]\!]^\rho(\sigma^\rho_2)$. This means that results obtained on σ^ρ can be used on $\sigma^\rho_1 \leq \sigma^\rho$.

In the following, we compute dependencies with respect to σ^ρ_n, which is the abstract state computed at n as an invariant at that program point. In the worst case, no information about the different variables is available, and σ^ρ_n associates [**top**] to all variables. Now, we aim at proving that e is independent from a set of variable X; in order to prove this, we need to prove that evaluating e always yields the same atom in ρ, independently from the value of variables in X, possibly without trying all possible values. Our idea is to prove this atomicity, if it holds, by iteratively refining (Section 2.4) the starting abstract state σ^ρ_n. The following example gives the intuition.

Example 6.8. Let $e \equiv \mathsf{x}{*}\mathsf{x} + 1$ and $\rho = \rho_{\text{SIGN}}$. Let also $[\![e]\!]^\rho$ follow the usual rules on $*$ and $+$: $[\mathbf{pos}] * [\mathbf{pos}] = [\mathbf{pos}]$, $[\mathbf{neg}] * [\mathbf{neg}] = [\mathbf{pos}]$, $[\mathbf{top}] * [\mathbf{top}] = [\mathbf{top}]$, $[\mathbf{pos}] + [\mathbf{pos}] = [\mathbf{pos}]$, $[\mathbf{top}] + [\mathbf{pos}] = [\mathbf{top}]$, etc. Suppose to start from a memory such that $\{\mathsf{x} \leftarrow [\mathbf{top}]\}$; we observe that $[\![e]\!]^\rho(\{\mathsf{x} \leftarrow [\mathbf{top}]\}) \in \textsc{Atoms}(\rho_{\text{SIGN}})$ cannot be proved by using these rules, since $[\mathbf{top}] * [\mathbf{top}] + [\mathbf{pos}] = [\mathbf{top}]$, which is not atomic, so that there may be a dependency. On the other hand, consider the possible refinements w.r.t. x in ρ_{SIGN}, namely, $\{\{\mathsf{x} \leftarrow [\mathbf{pos}]\}, \{\mathsf{x} \leftarrow [\mathbf{zero}]\}, \{\mathsf{x} \leftarrow [\mathbf{neg}]\}\}$. These abstract values are a covering (Section 2.4) of $\{\mathsf{x} \leftarrow [\mathbf{top}]\}$ since all possible values of x are accounted for by the refinements "x is positive", "x is zero", "x is negative". Moreover, it turns out that

$$[\![e]\!]^\rho(\{\mathsf{x} \leftarrow [\mathbf{pos}]\}) = [\![e]\!]^\rho(\{\mathsf{x} \leftarrow [\mathbf{zero}]\}) = [\![e]\!]^\rho(\{\mathsf{x} \leftarrow [\mathbf{neg}]\}) = [\mathbf{pos}]$$

Thus, the positivity of x can be proved by using a kind of *case-based reasoning* which requires computing e three times (plus the initial one on $\{\mathsf{x} \leftarrow [\mathbf{top}]\}$) instead of just once.

[7] Remember that atoms in ρ are [**zero**], [**poseven**], [**posodd**], [**negeven**], and [**negodd**]; since this is a partition of concrete values, we describe all concrete inputs by computing $[\![e]\!]^\rho$ on atoms.

[8] Since *Atom-dep* is just an over-approximation of semantic dependencies, it can be the case that such different results actually do not exist; on the other hand, atomicity does prove that no such results exist.

Given an initial abstract state σ^ρ, we define the set of all abstract states that "refine" abstract values on variables outside X to atomic values, while leaving X untouched.

$$[\sigma^\rho|X] = \left\{ \ \sigma_i^\rho \ \middle| \ \forall y \notin X.\, \sigma_i^\rho(y) \in \textsc{Atoms}\,(\rho) \ \ \wedge \ \ \forall x \in X.\, \sigma_i^\rho(x) = \sigma^\rho(x) \ \right\}$$

To prove e independent from x, we need to prove $\textsc{Atom}_\rho\left(\left[\!\left[e\right]\!\right]^\rho(\sigma_x^\rho)\right)$ for any $\sigma_x^\rho \in [\sigma_n^\rho|\{x\}]$. This amounts to say that any variation in x (up to the invariant $\sigma_n^\rho(x)$) does not lead to an observable variation in e, whenever all the other variables are fully specified as atoms. The following definition is based on the notion of covering introduced in Section 2.4, and identifies how an abstract state σ^ρ can be incrementally "refined", i.e., how it can be replaced by a set of states which is a covering, and where values of variables outside some set X take values which are either the same of the corresponding values in σ^ρ, or their direct sub-values.

$$X\text{-}\mathit{covering}(\sigma^\rho) = \left\{ \{\sigma_1^\rho, \ldots, \sigma_k^\rho\} \,\middle|\, \begin{array}{l} \{\sigma_1^\rho, \ldots, \sigma_k^\rho\} \text{ is a covering and} \\ \forall i.\, \forall x \in X. \sigma_i^\rho(x) = \sigma^\rho(x) \ \wedge \ \forall y \notin X.\, \sigma_i^\rho(y) \leq_\iota \sigma^\rho(x) \end{array} \right\}$$

where $\sigma_1^\rho(x) \leq_\iota \sigma_2^\rho(x)$ iff $\sigma_1^\rho(x) = \sigma_2^\rho(x)$ or $\sigma_1^\rho(x)$ is a *direct* sub-value of $\sigma_2^\rho(x)$ in ρ (for example, in $\rho_{\textsc{parSign}}$, [**poseven**] is a direct sub-value of [**pos**] but [**bot**] is not because there is another abstract value between them). It is easy to see that, by repeatedly computing X-coverings starting from σ^ρ, one ends up generating all abstract states in $[\sigma^\rho|X]$. This is what the algorithm presented in the following does incrementally.

Example 6.9. Consider $\rho_{\textsc{parSign}}$ and the abstract state $\sigma^\rho = \{x \leftarrow [\mathbf{pos}], y \leftarrow [\mathbf{odd}], z \leftarrow [\mathbf{top}]\}$: the following set of states is a $\{z\}$-covering of σ^ρ.

$$\left\{ \begin{array}{c} \{x \leftarrow [\mathbf{poseven}], y \leftarrow [\mathbf{odd}], z \leftarrow [\mathbf{top}]\}, \\ \{x \leftarrow [\mathbf{posodd}], y \leftarrow [\mathbf{odd}], z \leftarrow [\mathbf{top}]\} \end{array} \right\}$$

Another $\{z\}$-covering is the following, which refines both x and y:

$$\left\{ \begin{array}{c} \{x \leftarrow [\mathbf{poseven}], y \leftarrow [\mathbf{posodd}], z \leftarrow [\mathbf{top}], \} \\ \{x \leftarrow [\mathbf{poseven}], y \leftarrow [\mathbf{negodd}], z \leftarrow [\mathbf{top}], \} \\ \{x \leftarrow [\mathbf{posodd}], y \leftarrow [\mathbf{posodd}], z \leftarrow [\mathbf{top}], \} \\ \{x \leftarrow [\mathbf{posodd}], y \leftarrow [\mathbf{negodd}], z \leftarrow [\mathbf{top}]\} \end{array} \right\}$$

Moreover, given an expression e and a set of variables X, we define

$$\begin{array}{rcl} \mathbb{A}_e^{\mathrm{U}}(\sigma^\rho) & \text{iff} & \left[\!\left[e\right]\!\right]^\rho(\sigma^\rho) = \mathrm{U} \wedge \textsc{Atom}_\rho(\mathrm{U}) \ \text{ or} \\ & & \exists \{\sigma^\rho{}_1, ..\sigma^\rho{}_k\} \text{ covering of } \sigma^\rho \text{ such that } \forall i.\ \mathbb{A}_e^{\mathrm{U}}(\sigma^\rho{}_i) \\ \mathbb{A}'_e(\sigma^\rho, X) & \text{iff} & (1)\ \ \mathbb{A}_e^{\mathrm{U}}(\sigma^\rho) \ \text{ for some atom } \mathrm{U} \text{ or} \\ & & (2)\ \ \exists \{\sigma_i^\rho\}_{i \in [1,k]} \in X\text{-}\mathit{covering}(\sigma^\rho).\ \forall i.\ \mathbb{A}'_e(\sigma_i^\rho, X) \end{array}$$

The predicate $\mathbb{A}_e^{\mathrm{U}}(\sigma^\rho)$ holds if the expression e can be proved to have an atomic value U, either by direct computation of the abstract semantics, or by case-based reasoning like in Example 6.8, relying on the notion of covering. On the other hand, $\mathbb{A}'_e(\sigma^\rho, X)$ holds if it can be proven that there is no dependency of e on variables in X. This can be proven (1) directly, if e has an atomic value in σ^ρ (which implies that there is no dependency on variables in X or any other variables; or (2) recursively, by replacing σ^ρ with an X-covering, and trying to prove the result for every refinement. Condition (2) relies on case-based reasoning (which is similar to the one outlined in Example 6.8, but not to be confused with it), and allows to indirectly prove non-dependency.

Example 6.10. Consider the expression e =2∗x+y, the $\rho_{\textsc{parSign}}$ domain and the initial abstract state $\sigma_n^\rho = \{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{top}]\}$ (i.e., it is already known that x is positive). The predicate $\mathbb{A}'_e(\sigma_n^\rho, \{\mathsf{x}\})$ does not hold directly by condition (1) since the value of e in σ_n^ρ is

obviously not atomic, and there does not exist any covering which allows to prove $\mathbb{A}_e^{\mathrm{U}}(\sigma_n^\rho)$ for some U. Then, $\{\mathsf{x}\}$-coverings have to be searched for, which allow to prove condition (2). One such $\{\mathsf{x}\}$-covering is (the other $\{\mathsf{x}\}$-coverings are not treated explicitly)

$$\{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{even}]\}, \{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{odd}]\}$$

Condition (2) would be proved if both $\mathbb{A}'_e(\{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{even}]\}, \{\mathsf{x}\})$ and $\mathbb{A}'_e(\{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{odd}]\}, \{\mathsf{x}\})$ hold. However, none of them hold: for example, the abstract evaluation of $[\mathbf{poseven}] * [\mathbf{pos}] + [\mathbf{even}]$ gives the non-atomic value $[\mathbf{even}]$. Therefore, it is necessary to perform another recursive step: an $\{\mathsf{x}\}$-covering of $\{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{even}]\}$ would be, for example,

$$\{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{poseven}]\}, \{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{negeven}]\}, \{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{zero}]\}$$

Unfortunately, it is still not possible to prove $\mathbb{A}'_e(\sigma^\rho{}_i, \{\mathsf{x}\})$ for all three abstract states in the $\{\mathsf{x}\}$-covering, and they cannot be further refined because values for y are already atomic. Therefore, non-dependency of e on x cannot be proved. For the sake of simplicity, this example does not consider explicitly all $\{\mathsf{x}\}$-coverings, but the result we give is true: $\mathbb{A}'_e(\sigma^\rho, \{\mathsf{x}\})$ does not hold.

Example 6.11. Consider the same expression $e = 2{*}\mathsf{x}{+}\mathsf{y}$, the ρ_{PARSIGN} domain and the initial abstract state $\sigma_n^\rho = \{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{pos}]\}$. In this case, it is possible to prove $\mathbb{A}'_e(\sigma_n^\rho, \{\mathsf{x}\})$ because there is an $\{\mathsf{x}\}$-covering

$$\{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{poseven}]\}, \{\mathsf{x} \leftarrow [\mathbf{pos}], \mathsf{y} \leftarrow [\mathbf{posodd}]\}$$

which satisfies condition (2). Therefore, if it is known that both x and y are positive, then it is possible to prove the independency of e from x.

PROPOSITION 6.12 (SOUNDNESS). *Let $\rho \in uco(\wp(\mathbb{V}))$, and let $\overline{\rho}$ denote the tuple of ρ, on each variable. If $\mathbb{A}'_e(\sigma_n^\rho, X)$ can be proved, then there is no $x \in X$ such that $x \overset{\overline{\rho}, \rho}{\rightsquigarrow}_{\text{AT}} e$.*

PROOF. By induction. Consider the definition of $\mathbb{A}'_e$:

— If $\mathbb{A}_e^{\mathrm{U}}(\sigma_n^\rho)$ holds for some atomic abstract value U, then e depends on no variables.
— Otherwise, let $\{\sigma_1^\rho, .., \sigma_k^\rho\}$ be an X-covering of σ_n^ρ; by inductive hypothesis, for every i, e can be proved not to depend on X if values are limited by σ_i^ρ. Since these states are also a (regular) covering of σ_n^ρ, and variables in X are never restricted in any of them (i.e., all values allowed by the initial σ_n^ρ are considered for every i), this means that e does not depend on X.

□

The FINDNDEPS algorithm depicted in Figure 14 starts by trying to prove $\mathbb{A}'_e(\sigma_n^\rho, \text{VARS}(e))$, where $\text{VARS}(e)$ are all the variables syntactically occurring in the expression. If it succeeds, then e depends on no variables. Otherwise, the variable set X is decreased non-deterministically (one element at a time, randomly) until some judgment $\mathbb{A}'_e(\sigma_n^\rho, X)$ is proved.

The soundness of the algorithm follows easily from Proposition 6.12. On the other hand, the algorithm is also complete with respect to *Atom-dep* (which obviously does not mean that it exactly computes semantic abstract dependencies, due to the fact that *Atom-dep* is already an over-approximation of *Ndep*). In fact, exploring incrementally the space of X-coverings of σ_n^ρ is a way to consider all states in $[\sigma_n^\rho | X]$ without the need to generate all of them explicitly. Indeed, if non-dependency from X can be already proved for an abstract state belonging to an X-covering, then there is no need to consider all its refinements belonging to $[\sigma^\rho | X]$.

```
1 function findNdeps(e, ρ, σ_n^ρ) {
2   nonDep := ∅; // can be modified by prove()
3   prove(e, σ_n^ρ, vars(e));
4   return vars(e) ∖ nonDep; // relevant variables
5 }
6 procedure prove(e, σ^ρ, X) {
7   if (𝔸'_e(σ^ρ, X) ≠ ⊥) then nonDep := nonDep ∪ X;
8   else foreach (x ∈ X) { prove(e, σ^ρ, X ∖ {x}); }
9 }
```

Fig. 14. The findNdeps algorithm

Importantly, the assertions $\mathbb{A}'_e(\sigma^\rho_n, X)$ and $\mathbb{A}'_e(\sigma^\rho_n, Y)$ guarantee $X \cup Y \not\rightsquigarrow_{\text{AT}} e$ not to hold, even if $\mathbb{A}'_e(\sigma^\rho_n, X \cup Y)$ cannot be directly proved. The final result of findNdeps is $\text{vars}(e) \setminus Z$, where Z is the union of all sets Z_i such that $\mathbb{A}'_e(\sigma^\rho_n, Z_i)$ can be proved. By Propositions 6.6 and 6.12, this set is an over-approximation of relevant variables.

The findNdeps algorithm may deal, in principle, with *infinite* abstract domains, in particular those with infinite descending chains, since non-dependency results can be possibly proved without exploring the entire state-space; in fact, as pointed our above, if $\mathbb{A}^{\text{U}}_e(\sigma^\rho_n)$ can be proved, then it is not needed to descend into the (possibly infinite) set of sub-states of σ^ρ_n. This is not possible in the brute-force approach. It is also straightforward to add *computational bounds* in order to stop refining states if some amount of computational effort has been reached.

6.2.2. Dependency erasure. The problem of computing abstract dependencies can be observed from another point of view: given e and a set X of variables, we may be interested in the *most concrete* ρ such that $X \not\rightsquigarrow e$ does not hold. This would be the most concrete observation guaranteeing the non-interference of the variables in X on the evaluation of e [Giacobazzi and Mastroeni 2004a; Mastroeni 2013]. This task can be accomplished by repeatedly simplifying an initial domain ρ_0 in order to eliminate abstract values which are *responsible* for dependencies. As discussed before, in order not to have dependencies on X, we should have $\mathbb{A}'_e(\sigma^\rho_n, X)$, i.e., $\mathbb{A}^{\text{U}}_e(\sigma^\rho_X)$ should hold for any $\sigma^\rho_X \in [\sigma^\rho_n | X]$. If this does not hold for some abstract state σ^ρ_X, then ρ is modified to make $[\![e]\!]^\rho(\sigma^\rho_X)$ atomic.

We design a simple algorithm $\text{Edep}(e, \rho_0, X)$ (Figure 15) which takes an input an expression, an initial uco and a set of variables. The algorithm uses a queue in order to store abstract states; initially, the queue only contains σ^ρ_n. When a state σ^ρ is extracted from the queue, Edep checks (line 6) if there exists U atomic such that $\mathbb{A}^{\text{U}}_e(\sigma^\rho)$. If it is not, then

— If it is not possible to further refine σ^ρ (guard at line 7), then the abstract domain must be simplified (see below), and the algorithm re-starts from σ^ρ_n with the new domain.
— Otherwise, σ^ρ is refined according to the definition of X-covering (lines 13–15), and the process is repeated on all such refinements.

Each iteration of the loop corresponds to a value of the variable ρ. If the algorithm never reaches line 10 during a whole execution of a loop iteration (i.e., if, for every state σ^ρ, the condition $\mathbb{A}^{\text{U}}_e(\sigma^\rho)$ can be proved at line 6 before σ^ρ becomes "non-refinable" at line 7), then ρ is not simplified, which means that $\mathbb{A}'_e(\sigma^\rho_n, X)$ could be proved, thus demonstrating the non-dependency of e on X with respect to ρ. In fact, the goal of lines 6–15 is exactly to decide if $\mathbb{A}'_e(\sigma^\rho_n, X)$, either directly (line 6) or considering X-coverings.

The simplifying operator $\text{atomize}(\rho, \text{V})$ is a *domain transformer*, and works by removing abstract values in order to obtain $\text{Atom}_\rho(\text{V})$. Formally,

$$\rho' \;=\; \text{atomize}(\rho, \text{V}) \;\stackrel{\text{def}}{=}\; \{\text{U} \in \rho \mid \text{V} \sqcap \text{U} = \bot \;\vee\; \text{V} \leq \text{U}\}$$

```
1  ρ := ρ0 ; // the initial domain
2  repeat {
3    inputQueue := [σ_n^ρ] ; // one-element queue
4    while (notEmpty(inputQueue)) {
5      σ^ρ := extract(inputQueue) ;
6      if (∄U. A_e^U(σ^ρ)) then {
7        if (all vars in VARS(e) ∖ X are atomic in σ^ρ) then {
8          // at this point, V is not atomic
9          V := [[e]]^ρ(σ^ρ) ;
10         ρ := ATOMIZE(ρ, V) ;
11         // the queue is set again to a one-element queue
12         inputQueue := [σ_n^ρ] ;
13       } else { // {σ_1^ρ..σ_k^ρ} is an X-covering of σ^ρ
14         foreach(i) {
15           insertInQueue(inputQueue, σ_i^ρ) }}}}}
16 } until (ρ has not been modified in the whole loop iteration) ;
17 return ρ; // the domain s.t. X ⇝ e does not hold
```

Fig. 15. The EDEP algorithm.

Example 6.13. Consider ρ_{PARSIGN}: in order to make [**even**] be atomic, it is necessary to remove abstract values which are not disjoint, greater than or equal to [**even**]. In this case, all values but [**odd**] have to be removed from the abstract domain, so that $\text{ATOMIZE}(\rho_{\text{PARSIGN}}, [\mathbf{even}]) = \rho_{\text{PAR}}$.

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

THEOREM 6.14. *$\rho \overset{def}{=} \text{EDEP}(e, \rho_0, X)$ makes e not* Atom-dep *on X, that is, for every σ^ρ which is atomic on $\text{VARS}(e) \setminus X$, $[\![e]\!]^\rho(\sigma^\rho)$ is atomic. Moreover, by Proposition 6.6, e is not dependent on X with respect to Definition 6.2.*

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

Example 6.15. Consider the expression $e =$ 2∗x+y, the ρ_{PARSIGN} domain and the set of variables {x}. It is clear that e does depend on x with respect to ρ_{PARSIGN}. Suppose that the algorithm starts from $\sigma_n^\rho = \{\mathsf{x} \leftarrow [\mathbf{top}], \mathsf{y} \leftarrow [\mathbf{top}]\}$. Since $[\![e]\!]^\rho(\sigma_n^\rho) = [\mathbf{top}]$ and the condition at line 7 is false, an {x}-covering of σ_n^ρ is to be considered (line 15):

$$\{\mathsf{x} \leftarrow [\mathbf{top}], \mathsf{y} \leftarrow [\mathbf{even}]\}, \quad \{\mathsf{x} \leftarrow [\mathbf{top}], \mathsf{y} \leftarrow [\mathbf{odd}]\}$$

For each abstract state in the $\{x\}$-covering the same test is performed, but the value of the expression is still non-atomic. The computation goes on without computing atomic values for e, until the guard at line 7 becomes *true* (for example, when considering $\{\mathsf{x} \leftarrow [\mathbf{top}], \mathsf{y} \leftarrow [\mathbf{poseven}]\}$). Since the abstract value computed in this state is [**even**], the procedure $\text{ATOMIZE}(\rho_{\text{PARSIGN}}, [\mathbf{even}])$ is called at line 10, and the new candidate abstract domain ρ_{PAR} is computed (Example 6.13). Next, the queue is reset to its initial value, and another iteration of the loop begins. By following the same reasoning, it is easy to see that the dependency no longer holds in the new domain ρ_{PAR}, so that the second iteration of the loop does not simplify the domain anymore, and ρ_{PAR} is finally returned.

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 $\mathbb{A}_e$ 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 end of Section 6.2.1). Moreover, unlike FINDNDEPS, there is no reasonable trivial counterpart, since any brute-force approach would be definitely impractical.

7. THE QUEST FOR ABSTRACT SLICES

This section introduces an algorithm for computing conditioned abstract slices, based on abstract dependencies and the notion of *agreement* between states.

As explained in Section 7.4, this way to compute slices relies on *a priori* knowledge of the properties which will be of interest for the analysis. In most cases, the majority of the abstract domains to be taken into account are quite simple (e.g., nullity). On the other hand, Section 6.2 presents a general way to compute abstract dependencies on more complex domains. Indeed, those algorithms can be used here, and their complexity is acceptable if small domains are dealt with.

The slices obtained by following this methodology will have the standard form of backward slicing; *predicates* or *conditions* β on states are supported, in the style of conditioned slicing [Canfora et al. 1998]. In the rules, the judgment $\sigma \models \beta$ means that the state σ satisfies β. It must be pointed out that a predicate β at a certain program point may include user-provided information but also knowledge about program variable which is obtained by performing static analysis on the program (see the discussion in Section 2.4 and the use of σ_n^ρ in Section 6.2). For example, after a `x:=new C()` statement, judgments like "`x` is not null", "`x` is not cyclic", "`x` is not sharing with `y`" or "`x` is not reaching `y`" could be provided, depending on the kind of static analyses available (nullity, sharing, cyclicity, etc.).

A way to decide which statements have to be included in an abstract slice consists of two main steps:

— for each statement s, a specific static-analysis algorithm provides information about the relevant data *after* that statement (below, the *agreement*), according to the slicing criterion and the program code;
— if the execution of s *does not affect* its corresponding agreement (i.e., some condition on states which must hold after s), then s can be removed from the slice.

Example 7.1. Consider the following code fragment, and suppose that the slicing criterion is the nullity of `x` at the end:

```
21  ...
22  y.f := exp;
23  x := y;
```

The field update on `y` (line 22) can be removed from the slice because

— the question about the nullity of `x` after line 23 is equivalent to the question about the nullity of `y` after line 22; and
— the field update at line 22 does not affect the nullity of `y`.

The rest of this section formalizes how these two main steps are carried out.

7.1. The logic for propagating agreements

This section describes how agreements are defined and propagated via a system of logical rules: the G-system[9]. Hoare-style *triples* [Hoare 1969] are used for this purpose, in the spirit of the *weakest precondition calculus* [Dijkstra 1975].

DEFINITION 7.2. *An* agreement $\mathcal{G}$ *is a set of conditions* $[\bar{x}::\rho]$ *where each uco* ρ *involves a sequence of variables* $\bar{x}$ *(most usually, just one variable), and all conditions involve mutually disjoint sets of variables. Two states* σ_1 *and* σ_2 *are said to* agree *on* $\mathcal{G}$*, written* $\mathcal{G}(\sigma_1, \sigma_2)$*, iff, for every* $[\bar{x}::\rho]$ *in* $\mathcal{G}$*,* $\rho(\sigma_1(\bar{x})) = \rho(\sigma_2(\bar{x}))$*, where notation is abused by taking* $\sigma_i(\bar{x})$ *as the sequence of values of variables* $\bar{x}$ *in* σ_i*, and* $\rho(\sigma_i(\bar{x}))$ *as the application of* ρ *(which could be a relational domain) to the elements of such a sequence.*

Agreements are easily found to form a lattice, and a partial order $\sqsubseteq$ *can be defined:* $\mathcal{G}' \leq \mathcal{G}''$ *iff, for every* σ_1*,* σ_2 *such that* $\mathcal{G}'(\sigma_1, \sigma_2)$*, then* $\mathcal{G}''(\sigma_1, \sigma_2)$*. Moreover, an intersection operator is induced by the partial order:* $\mathcal{G}' \sqcap \mathcal{G}''$ *is the greatest agreement which is less than or equal to both.*

In the following, $\mathcal{G}(x)$ will be the uco corresponding to the condition $[x::\rho]$ in $\mathcal{G}$, or $\rho_\top$ if no condition on x belongs to $\mathcal{G}$. For the sake of simplicity, the discussion will be limited to domains each involving one single variable. In this case, ordering amounts to the following: $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ if $\forall x.\mathcal{G}_1(x) \sqsubseteq \mathcal{G}_2(x)$, where $\mathcal{G}_1(x) \sqsubseteq \mathcal{G}_2(x)$ is the comparison on the *precision* of abstract domains, meaning that $\mathcal{G}_1(x)$ is *more precise* than $\mathcal{G}_2(x)$.

Example 7.3. Let $\sigma_1 = \{\mathsf{n} \leftarrow 2, \mathsf{i} \leftarrow 3, \mathsf{x} \leftarrow \textbf{null}\}$ and $\sigma_2 = \{\mathsf{n} \leftarrow 0, \mathsf{i} \leftarrow 4, \mathsf{x} \leftarrow \textbf{null}\}$ be two states. Then, they agree on $\mathcal{G} = \{[\langle \mathsf{n} \rangle::\rho_{\text{PAR}}], [\langle \mathsf{x} \rangle::\rho_{\text{NULL}}]\}$ since

— n has the same parity in both states;
— x is **null** (therefore, it has "the same nullity") in both states; and
— there is no condition on i.

On the other hand, these states do not agree on $\mathcal{G}' = \{[\langle \mathsf{i} \rangle::\rho_{\text{PAR}}], [\langle \mathsf{x} \rangle::\rho_{\text{NULL}}]\}$ because i is odd in σ_1 whereas it is even in σ_2.

In a triple $\{\mathcal{G}\}\ s\ \{\mathcal{G}'\}$, the pre-condition $\mathcal{G}$ is the weakest agreement on two states before a statement s such that the agreement specified by the post-condition $\mathcal{G}'$ holds after the statement. Predicates on states can be used, so that triples are, actually, 4-tuples which only take into account a subset of the states. Formally, the 4-tuple (or *augmented triple*) $\{\mathcal{G}\}^\beta\ s\ \{\mathcal{G}'\}$ (where the *true* predicate is often omitted) holds if, for every σ_1 and σ_2,

$$\sigma_1 \models \beta \ \wedge\ \sigma_2 \models \beta \ \wedge\ \mathcal{G}(\sigma_1, \sigma_2) \Rightarrow \mathcal{G}'([\![s]\!](\sigma_1), [\![s]\!](\sigma_2))$$

The rules of the G-system are shown in Figure 16. The *transformed predicate* $s(\beta)$ is one which is guaranteed to hold after a statement s, given that β holds before, in the style of *strongest post-condition calculus* [Dijkstra and Scholten 1990; Bijlsma and Nederpelt 1998]. For example, if $\beta = x \geq 0$, then the condition $s(\beta) = x \geq 1 \wedge y \neq \textbf{null}$ certainly holds after $s \equiv$ x:=x+1; y:=**new** C(). The way predicates are transformed is outside the scope of this paper; however, the cited works introduced calculi for computing such strongest post-conditions. In the absence of such tools, *true* is always a consistent choice (precision, but not soundness, may be affected since the set of states to be considered grows larger, and to prove useful results could become harder).

The G-system is tightly related to narrow non-interference [Giacobazzi and Mastroeni 2004a; 2004b]. These works define a similar system of rules, the N-rules, for assertions

[9] A version of this system was introduced in previous work [Zanardini 2008] as the A-system. However, there are many differences between both systems, mainly due to the changes in the language under study (for example, variables are taken into account here instead of a more involved notion of *pointer expression*).

$[\eta]\, s\, (\eta')$, where η and η' are basically the (tuples of) abstract domains corresponding to, resp., $\mathcal{G}$ and $\mathcal{G}'$. The systems differ in that:

— the use of pointers requires the rules for assignment to account for sharing, while N-rules only work on integers;
— in the present approach, domains are supposed to be partitioning, so that there is no need to include explicitly the Π operator (Section 2.3);
— the G-system does not distinguish between *public* and *private* since this notion is not relevant in slicing;
— the rule for conditional is not included in the N-system; indeed, this is quite a tricky rule, and, in general, expressing a conditional with loops and using the rule N6 for loops results in inferring less precise assertions;
— in the N-system, predicates β on program states are not supported.

In the following, each rule of the G-system is discussed. Importantly, this rule system relies on the computation of *property preservation*. We rely on a rule system which soundly computes whether executing a statement affects properties of some variables: the judgment

$$\text{PP}^{\beta}\,(\mathcal{G}, s)$$

can only be obtained by using those rules if it is possible to prove that executing s in a state σ satisfying the condition β results in a final state σ' that is equal to the initial σ with respect to the agreement $\mathcal{G}$, i.e., $\mathcal{G}(\sigma, \sigma')$. In other word, the statement is equivalent to **skip** with respect to the properties of interest. Note that, here, the agreement is not used to compare two states at the same program point; rather, it takes as input the states before and after executing a statement. The rule system for proving property preservation is explained after introducing the G-system (Section 7.1.7).

7.1.1. Rule G-PP. This rule makes the relation between property preservation and the G-system more clear. The triple $\{\mathcal{G}\}\ s\ \{\mathcal{G}\}$ amounts to say that two executions agree *after* s, provided they agree *before* on the same $\mathcal{G}$. On the other hand, the preservation of $\mathcal{G}$ on s means that any state *before* s agrees on $\mathcal{G}$ with the corresponding state *after* s. Property preservation is a stronger requirement than the mere propagation $\{\mathcal{G}\}\ s\ \{\mathcal{G}\}$ of agreements, so that this rule is sound. In fact, if $\mathcal{G}(\sigma_1, \sigma_2)$ and both $\mathcal{G}(\sigma_1, [\![s]\!]\,(\sigma_1))$ and $\mathcal{G}(\sigma_2, [\![s]\!]\,(\sigma_2))$ hold, then $\mathcal{G}([\![s]\!]\,(state_1), [\![s]\!]\,(\sigma_2))$ follows, which is, equivalent, by definition, to $\{\mathcal{G}\}\ s\ \{\mathcal{G}\}$.

Example 7.4. Let the parity of x be the property of interest. In this case, x := x+1 does not preserve the parity of x, but two initial states agreeing on $[x{::}\rho_{\text{PAR}}]$ lead to final states which still agree on it. Therefore, $\{[x{::}\rho_{\text{PAR}}]\}$ x := x+1 $\{[x{::}\rho_{\text{PAR}}]\}$ holds. On the other hand, x := x+2 also satisfies a stronger requirement: that $\rho_{\text{PAR}}(x)$ does not change. Therefore, besides having $\{[x{::}\rho_{\text{PAR}}]\}$ x := x+2 $\{[x{::}\rho_{\text{PAR}}]\}$, the judgment PP $([x{::}\rho_{\text{PAR}}]$, x := x+2) is also true.

7.1.2. Rules G-SKIP, G-CONCAT, G-ID, G-SUB. The G-SKIP rule describes *no-op*. The assertion holds for every $\mathcal{G}$ and β since $[\![\textbf{skip}]\!]\,(\sigma) = \sigma$.

G-CONCAT is also easy: soundness holds by transitivity (note also the use of $s(\beta)$ to propagate conditions of states).

Rule G-ID can be used when nothing else can be proved: it always holds because execution is deterministic, so that two executions starting from two states which are equal[10] on all variables will end in a pair of states agreeing on any abstraction. Note that, as pointed out in Section 2.1, **read** statements are supposed only to appear at the beginning of a program; therefore, a compound statement never contains any **read**, and the propagation of agreements can be done safely until reaching the first (actually, the last occurrence in

[10]The notion of equality on references and objects is recalled in Example 2.2.

$$\frac{\mathrm{PP}^{\beta}(\mathcal{G}, s)}{\{\mathcal{G}\}^{\beta}\ \ s\ \ \{\mathcal{G}\}}\ \text{G-PP} \qquad \frac{}{\{\mathcal{G}\}^{\beta}\ \ \textbf{skip}\ \ \{\mathcal{G}\}}\ \text{G-SKIP}$$

$$\frac{\{\mathcal{G}\}^{\beta}\ \ s\ \ \{\mathcal{G}'\} \qquad \{\mathcal{G}'\}^{s(\beta)}\ \ s'\ \ \{\mathcal{G}''\}}{\{\mathcal{G}\}^{\beta}\ \ s;s'\ \ \{\mathcal{G}''\}}\ \text{G-CONCAT} \qquad \frac{\forall x.\ \mathcal{G}_{ID}(x)=\rho_{\mathrm{ID}} \quad s \neq \textbf{read}(\cdot)}{\{\mathcal{G}_{ID}\}^{\beta}\ \ s\ \ \{\mathcal{G}'\}}\ \text{G-ID}$$

$$\frac{\{\mathcal{G}_2\}^{\beta_2}\ \ s\ \ \{\mathcal{G}_2'\} \qquad \mathcal{G}_1 \sqsubseteq \mathcal{G}_2 \qquad \mathcal{G}_2' \sqsubseteq \mathcal{G}_1' \qquad \beta_1 \Rightarrow \beta_2}{\{\mathcal{G}_1\}^{\beta_1}\ \ s\ \ \{\mathcal{G}_1'\}}\ \text{G-SUB}$$

$$\frac{\forall y.\ \neg\left(y\ \overset{\mathcal{G},\mathcal{G}'(x)}{\rightsquigarrow}_{\mathrm{AT}}\ e\right)^{\beta} \qquad \forall y \neq x.\ \mathcal{G}(y) = \mathcal{G}'(y)}{\{\mathcal{G}\}^{\beta}\ \ x{:=}e\ \ \{\mathcal{G}'\}}\ \text{G-ASSIGN}$$

$$\frac{\begin{array}{ll}(*) & \forall y \in \mathrm{DAL}(x)\,.\ \forall \sigma_1 \models \beta, \sigma_2 \models \beta.\\ & \quad \mathcal{G}(\sigma_1,\sigma_2) \Rightarrow\\ & \quad \mathcal{G}'(\sigma_1[y.f \leftarrow [\![e]\!](\sigma_1)], \sigma_2[y.f \leftarrow [\![e]\!](\sigma_2)])\\ (**) & \forall y \in \mathrm{SH}(x)\,.\ \forall \bar{g}, \forall \sigma_1 \models \beta, \sigma_2 \models \beta.\\ & \quad \mathcal{G}(\sigma_1,\sigma_2) \Rightarrow\\ & \quad \mathcal{G}'(\sigma_1[y.\bar{g} \leftarrow [\![e]\!](\sigma_1)], \sigma_2[y.\bar{g} \leftarrow [\![e]\!](\sigma_2)])\\ (***) & \forall y \notin \mathrm{DAL}(x)\,.\ \mathcal{G}(y) \sqsubseteq \mathcal{G}'(y)\end{array}}{\{\mathcal{G}\}^{\beta}\ \ x.f{:=}e\ \ \{\mathcal{G}'\}}\ \text{G-FASSIGN}$$

$$\frac{\{\mathcal{G}\}^{\beta}\ \ s_t \diamond s_f\ \ \{\mathcal{G}'\}}{\{\mathcal{G}\}^{\beta}\ \ \textbf{if}\ \ (b)\ s_t\ \textbf{else}\ s_f\ \ \{\mathcal{G}'\}}\ \text{G-IF1}$$

$$\frac{\{\mathcal{G}_t\}^{\beta\wedge b}\ \ s_t\ \ \{\mathcal{G}'\} \qquad \{\mathcal{G}_f\}^{\beta\wedge\neg b}\ \ s_f\ \ \{\mathcal{G}'\}}{\{\mathcal{G}_b \sqcap \mathcal{G}_t \sqcap \mathcal{G}_f\}^{\beta}\ \ \textbf{if}\ \ (b)\ s_t\ \textbf{else}\ s_f\ \ \{\mathcal{G}'\}}\ \text{G-IF2}$$

$$\frac{s(\beta) \Rightarrow \beta \quad \{\mathcal{G} \sqcap \mathcal{G}_b\}^{\beta\wedge b}\ \ s\ \ \{\mathcal{G} \sqcap \mathcal{G}_b\}}{\{\mathcal{G} \sqcap \mathcal{G}_b\}^{\beta}\ \ \textbf{while}\ \ (b)\ s\ \ \{\mathcal{G} \sqcap \mathcal{G}_b\}}\ \text{G-WHILE}$$

Fig. 16. The G-system

the code) **read** statement (in presence of read statements, there is no need to propagate agreements until the very first line of the code).

Finally, in G-SUB, remember that $\sqsubseteq$ is the partial order on agreements.

7.1.3. Rule G-ASSIGN. This rule means that, given a statement $x{:=}e$, any agreement $\mathcal{G}$ which satisfies the two conditions of the above part of the rule is a sound pre-condition for the post-condition $\mathcal{G}'$. The conditions are (1) that, given two states which agree on $\mathcal{G}$, the computed results for the expression e in both states are abstracted by $\mathcal{G}'(x)$ to the same abstract value; and (2) that $\mathcal{G}$ is as precise as $\mathcal{G}'$ on all variables but x. The first condition is represented in terms of Definition 6.5, and the superscript β indicates that only states satisfying β have to be considered. Such a condition can be easily shown to imply the formula

$$F \quad = \quad \forall \sigma_1 \models \beta, \sigma_2 \models \beta.\ (\mathcal{G}(\sigma_1,\sigma_2) \Rightarrow (\mathcal{G}'(x))([\![e]\!](\sigma_1)) = (\mathcal{G}'(x))([\![e]\!](\sigma_2)))$$

Note that, by Proposition 6.6, the absence of abstract dependencies w.r.t. Definition 6.5 (*Atom-dep*) implies the absence of abstract dependencies w.r.t. Definition 6.2 (*Ndep*) which, in turn, implies F. This clarifies the relation between abstract dependencies and the computation of a slice. Obviously, the second condition guarantees that, for all variables which are not updated by the assignment, the agreement required by $\mathcal{G}$ still holds when $\mathcal{G}'$ is considered.

7.1.4. Rule G-FASSIGN. This rule accounts for the modification of the data structure pointed to by a variable by means of a field update. In the following, that a variable is *affected* means that the data structure pointed to by it is updated. Given a field update on a variable x, some other variables (i.e., the data structures pointed to by them) could be affected. There exists a well-known static analysis which tries to detect which variables point to a data structure which is updated by a field update on x: this analysis is known as *sharing analysis* [Secci and Spoto 2005; Zanardini 2015], and usually comes as *possible-sharing* analysis, where the set of variables which *could* be affected by a field update is over-approximated. Moreover, *aliasing analysis* [Hind 2001] can be used in order to compute the set of variables pointing exactly (and directly) to the same location as x; in this case, *definite-aliasing* analysis makes sense, which under-approximates the set of variables which *certainly* alias with x. According to the result of these analyses, reference variables can be partitioned in three categories: (1) variables which certainly alias with x, so that they can be guaranteed to be updated in their field f; (2) variables not in (1) which could share with x, so that they could be affected by the update in many ways; and (3) variables which certainly do not share with x, so that they are unaffected by the update. Let $\text{SH}(x)$ be the set of variables possibly sharing with x before the update, and $\text{DAL}(x)$ be the set of variables definitely aliasing with x. In the absence of a definite-aliasing analysis, then $\text{DAL}(x)$ can be safely taken as $\{x\}$.

Example 7.5. Consider the following code fragment:

```
10 if (...) then { y.f := x; } else { y.f := z; }
11 w := x;
12 x.g := e;
```

Suppose that, initially, no variable is sharing with any other variable (i.e., there is no overlapping between data structures referred by different variables), and that the truth value of the boolean guard cannot be determined statically, so that both branches of the conditional statement have to be considered as possible executions. In this case, the sharing and aliasing information before line 12 is as follows:

$$\text{SH}(\mathsf{x}) = \{\mathsf{x}, \mathsf{y}, \mathsf{w}\} \qquad \text{DAL}(\mathsf{x}) = \{\mathsf{x}, \mathsf{w}\}$$

Note that every variable is aliasing with itself (none of them is **null**), w is certainly aliasing with x because of line 11, and y is possibly sharing with x (the actual sharing depends on the value of the guard).

The G-FASSIGN rule comes with three pre-conditions. Pre-condition $(*)$ only applies to variables in category (1): those definitely aliasing with x. Pre-condition $(**)$ applies to category (2), while pre-condition $(***)$ applies to categories (2) and (3).

— $(*)$ requires that updating the field f of the location pointed to by x (and all variables definitely aliasing with it) leads to an agreement on $\mathcal{G}'$, provided that the initial states agree on $\mathcal{G}$.

— (∗∗) is similar, but states that the agreements must hold for *every* sequence of field selectors $\bar{g}$[11], possibly the empty sequence. This is needed since, given that some y may share with x, it cannot be known which fields of y will be updated, and how. In practice, only the sequences of field selectors which are compatible with the class hierarchy of the program under study have to be considered, as shown in Example 7.6.
— (∗∗∗) applies to variables that could be unaffected by the update, i.e., variables in categories (2) and (3) (note that the conditions $y \in \text{SH}(x)$ in (∗∗) and $y \notin \text{DAL}(x)$ in (∗∗∗) are not mutually exclusive, so that variables in category (2) satisfy both). The relation between $\mathcal{G}$ and $\mathcal{G}'$ is clear in this case, as agreement on $\mathcal{G}$ must entail agreement on $\mathcal{G}'$.

Example 7.6 (*Infeasible sequences of field selectors*). In a Java-like language, a sequence like $\langle .\mathsf{g}.\mathsf{f}\rangle$ is not compatible with the following class hierarchy since the class of g is D whereas f is declared in C:

```
class C { D f; D g; }        class D { D h; }
```

Example 7.7. Consider the statement $s \equiv \mathsf{x}.\mathsf{f} := \mathsf{y}$. Let $\beta = \mathit{true}$ and the agreement $\mathcal{G}'$ after s be $\{[\mathsf{x}::\rho_{\text{NULL}}], [\mathsf{z}::\rho_{\text{NULL}}]\}$. Let also $\text{SH}(\mathsf{x})$ before s be $\{\mathsf{x}\}$. In this case, an agreement $\mathcal{G}$ which satisfies the judgment

$$\{\mathcal{G}\}^{\mathit{true}}\ {}_{\mathsf{x}.\mathsf{f}:=\mathsf{y}}\ \{\mathcal{G}'\}$$

can be the same $\{[\mathsf{x}::\rho_{\text{NULL}}], [\mathsf{z}::\rho_{\text{NULL}}]\}$ because

— z is unaffected by the update (it belongs to category (3)), so that $\mathcal{G}(\mathsf{z})$ must be at least as precise as $\mathcal{G}'(\mathsf{z})$;
— category (2) contains no variables; and
— category (1) only includes x itself, and the nullity of x is clearly unaffected by the update.

One may think that the universal quantification on field sequences in pre-condition (∗∗) results in an unacceptable loss of precision. Indeed, to require that all possible updates to possibly-sharing variables preserve the desired agreements seems to be too strict. However, there are a number of things to be considered:

— A closer look to the rule shows that there is no easier way to account for sharing if traditional sharing analysis is used.
— Example 7.7 shows that it is still possible to get meaningful results on domains working on pointer variables.
— The state of the art in static analysis of object-oriented languages indicates that abstract domains on pointers are likely to be quite simple (ρ_{NULL} being one of them).
— There is recent work [Zanardini 2015] introducing a more precise, *field-sensitive* sharing analysis which computes *how* variables share: this analysis is able to detect which *fields* are or are not involved in paths in the heap converging from two variables to a shared location. In order to keep the discussion as simple as possible, the definition of G-FASSIGN given in Figure 16 uses traditional sharing analysis. However, the impact of field-sensitive sharing analysis is discussed in Section 7.4, where a refined version of G-FASSIGN is given.

The domain of cyclicity introduced in Section 2.2 represents information about *data structures* in the heap, not only program variables. In this sense, field updates have to be regarded as potentially affecting the propagation of agreements.

[11] In the following, the notation $\langle .f_1.f_2.....f_n\rangle$ (starting with a dot is intentional) will be used to represents sequences of field selectors.

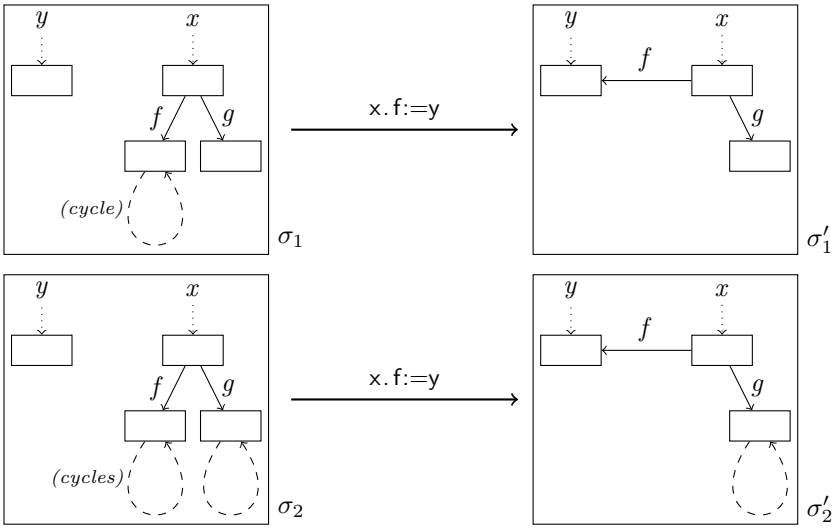


Fig. 17. How two executions agreeing on $\{[x::\rho_{\text{CYC}}], [y::\rho_{\text{CYC}}]\}$ do not agree on $\{[x::\rho_{\text{CYC}}]\}$ after x.f:=y

Example 7.8. Let s be, again, the statement x.f:=y, and $\mathcal{G}'$ be $\{[x::\rho_{\text{CYC}}]\}$; i.e., the interest is on the cyclicity of the data structure pointed to by x after executing s. Suppose also that x and y are certainly not sharing before s, and that an object whose type is compatible with x has two reference fields f and g. Then, the agreement $\mathcal{G} = \{[x::\rho_{\text{CYC}}], [y::\rho_{\text{CYC}}]\}$ is *not* a correct precondition for the Hoare tuple to hold, since there can be two states σ_1 and σ_2 (Figure 17) such that

— the data structure corresponding to y is acyclic, and equal in both states (therefore, there is an agreement on the cyclicity of y);
— the data structure corresponding to x is cyclic in both σ_1 and σ_2, but (a) in σ_1 there is only a cycle originating from the location bound to $x.f$; and (b) in σ_2 there is a cycle originating from $x.f$ and another one originating from $x.g$.

In this case, $\mathcal{G}(\sigma_1, \sigma_2)$ holds but the resulting final states σ'_1 and σ'_2 do *not* agree on $\mathcal{G}'$ since x is acyclic in σ'_1 (the only cycle has been broken) while it is still cyclic in σ'_2. This behavior is captured by G-FASSIGN because condition $(*)$ applied to x itself (which, by hypothesis, is the only variable in $\text{DAL}(x)$) does not hold, so that the augmented triple cannot be proven. On the other hand, $\{[x::\rho_{\text{ID}}], [y::\rho_{\text{CYC}}]\}$ would be a correct precondition.

7.1.5. Rules G-IF1 *and* G-IF2. In a conditional **if** (b) s_t **else** s_f there are two possibilities. Rule G-IF1 states that an input agreement which induces the output one whichever path is taken is a sound precondition. Here, the assertion $\{\mathcal{G}\}^{\beta}\ \ s_t \diamond s_f\ \ \{\mathcal{G}'\}$ means that

$$\forall \sigma_1, \sigma_2.\ \mathcal{G}(\sigma_1, \sigma_2) \wedge \sigma_1 \models \beta \wedge \sigma_2 \models \beta \Rightarrow \\ \mathcal{G}'([\![s_t]\!](\sigma_1), [\![s_t]\!](\sigma_2), [\![s_f]\!](\sigma_1), [\![s_f]\!](\sigma_2))$$

where the judgment $\mathcal{G}'(\cdot,\cdot,\cdot,\cdot)$ means that all four states agree on $\mathcal{G}'$. This rule requires $\mathcal{G}'$ to hold on the output state independently from the value of b. Soundness is easy (note that the above assertion implies $\{\mathcal{G}\}^{\beta}\ \ s_t\ \ \{\mathcal{G}'\}$ and $\{\mathcal{G}\}^{\beta}\ \ s_f\ \ \{\mathcal{G}'\}$).

Note that such a $\mathcal{G}$ can always be found (in the worst case, it assigns the identity upper closure operator ρ_{ID} to each variable, so that two states agree only if they are exactly equal). However, sometimes it can be more convenient to exploit information about b. In such cases,

G-IF2 can be applied, which means that the initial agreement $\mathcal{G}_t \sqcap \mathcal{G}_f$ is strong enough to verify the final one, provided the same branch is taken in both executions, as $\mathcal{G}_b$ requires. In fact, $\mathcal{G}_b$ is built from b, and separates states according to its value:

$$\mathcal{G}_b(\sigma_1, \sigma_2) \quad \Leftrightarrow \quad [\![b]\!](\sigma_1) = [\![b]\!](\sigma_2)$$

The rule means that, whenever two states agree on the branch to be executed, and the triples on both branches hold, the whole triple holds as well.

LEMMA 7.9 (SOUNDNESS OF G-IF2). *If σ_1 and σ_2 both satisfy β and agree on $\mathcal{G}_b \sqcap \mathcal{G}_t \sqcap \mathcal{G}_f$, then the corresponding output states σ_1' and σ_2' agree on $\mathcal{G}'$ under the hypotheses of the rule.*

PROOF. By hypothesis, the same branch is taken in both cases. Conditions $\beta \wedge b$ and $\beta \wedge \neg b$ are consistent since s_t (respectively, s_f) can only be executed when b is true (respectively, false). The agreement on $\mathcal{G}$ holds in both paths, so that the entire assertion is correct. □

Note that the rule to be chosen for the conditional depends on the precision of the outcome: G-IF2 can be a good choice if (1) it can be applied; and (2) the result is "better" than the one obtained by G-IF1. The second condition amounts to say that, given the same final agreement $\mathcal{G}'$, the initial agreement obtained by using G-IF2 is weaker (i.e., it is more likely that two states agree on it) than the one obtained by using G-IF1.

Example 7.10. Consider the code fragment

```
if (x>0) { x:=x+1; } else { x:=x−1; }
```

and let $\mathcal{G} = \{[x::\rho_{\text{SIGN}}]\}$ be the agreement after the statement, i.e., the relevant property is the sign of x. The rule G-IF2 is able to compute the same $\mathcal{G}$ as the input agreement because

— the triple $\{\mathcal{G}\}^{\beta \wedge b}\ \ s_t\ \ \{\mathcal{G}\}$ holds since the condition $\beta \wedge b$ guarantees that x is positive, and two states which agree on the sign before the increment will still agree after it (if x is positive in both states, then it will remain positive in both);
— similarly, the triple $\{\mathcal{G}\}^{\beta \wedge \neg b}\ \ s_f\ \ \{\mathcal{G}\}$ also holds (if x is 0 in both states, then it will be negative in both; and if it is negative in both, it will remain negative in both);
— $\mathcal{G}_b$ is less precise than $\mathcal{G}$ (i.e., $\mathcal{G} \sqsubseteq \mathcal{G}_b$ since the latter only separates numbers into positive and non-positive), so that the input agreement $\mathcal{G}_b \sqcap \mathcal{G} \sqcap \mathcal{G}$ is equal to $\mathcal{G}$.

On the other hand, G-IF1 is not able to compute the same input agreement because the precondition $\{\mathcal{G}\}^{\beta}\ \ s_t \diamond s_f\ \ \{\mathcal{G}\}$ of the rule does not hold. In fact, consider two states $\sigma_1 = \{\mathsf{x} \leftarrow 1\}$ and $\sigma_2 = \{\mathsf{x} \leftarrow 2\}$: they agree on the sign of x, but $([\![s_t]\!](\sigma_2))(\mathsf{x})$ and $([\![s_f]\!](\sigma_1))(\mathsf{x})$ have different sign (the first is zero while the second is positive).

7.1.6. Rule G-WHILE. The meaning of the rule for loops can be understood by discussing its soundness: if β is preserved after any iteration of the body, and the agreement which is preserved by the body guarantees the same number of iterations in both executions (i.e., it is more precise than $\mathcal{G}_b$), then such an agreement is preserved through the entire loop.

LEMMA 7.11 (SOUNDNESS OF G-WHILE). *Let σ_1^0 and σ_2^0 satisfy β, and agree on $\mathcal{G} \sqcap \mathcal{G}_b$. Then, given $\sigma_i' = [\![\textbf{while } (b)\ s_w]\!](\sigma_i^0)$, the result $(\mathcal{G} \sqcap \mathcal{G}_b)(\sigma_1', \sigma_2')$ holds.*

PROOF. Let $\sigma_i^{n+1} = [\![s_w]\!](\sigma_i^n)$ for every $n \geq 0$, i.e., σ_i^{n+1} refers to the situation after the $(n{+}1)$-th iteration. There are two cases, depending on the value of the boolean condition:

— $[\![b]\!](\sigma_1^n) = [\![b]\!](\sigma_2^n) = \mathit{false}$: in this case, the body is not executed, and the result holds trivially;
— $[\![b]\!](\sigma_1^n) = [\![b]\!](\sigma_2^n) = \mathit{true}$: in this case, $\sigma_i^n \models \beta \wedge b$. By the hypothesis of the rule, σ_1^{n+1} and σ_2^{n+1} agree on $\mathcal{G} \sqcap \mathcal{G}_b$, and β still holds since $s_w(\beta) \Rightarrow \beta$.

$$\frac{\mathrm{PP}^{\beta'}(\mathcal{G}', s) \quad \beta \Rightarrow \beta' \quad \mathcal{G}' \sqsubseteq \mathcal{G}}{\mathrm{PP}^{\beta}(\mathcal{G}, s)}\ \text{PP-WEAK} \qquad \frac{}{\mathrm{PP}^{\beta}(\mathcal{G}, \textbf{skip})}\ \text{PP-SKIP}$$

$$\frac{\mathrm{PP}^{\beta}(\mathcal{G}, e) \qquad \forall \sigma.\ \sigma \models \beta \Rightarrow \mathcal{G}(x)(\sigma(x)) = \mathcal{G}(x)([\![e]\!](\sigma))}{\mathrm{PP}^{\beta}(\mathcal{G}, x := e)}\ \text{PP-ASSIGN}$$

$$\frac{\begin{array}{l}(*)\ \ \forall y \in \mathrm{DAL}(x).\ \forall \sigma \models \beta.\ \mathcal{G}(\sigma, \sigma[y.f \leftarrow [\![e]\!](\sigma)]) \\ (**)\ \ \forall y \in \mathrm{SH}(x).\ \forall \bar{g}, \forall \sigma \models \beta.\ \mathcal{G}(\sigma, \sigma[y.\bar{g} \leftarrow [\![e]\!](\sigma)])\end{array}}{\mathrm{PP}^{\beta}(\mathcal{G}, x.f{:=}e)}\ \text{PP-FASSIGN}$$

$$\frac{\mathrm{PP}^{\beta}(\mathcal{G}, s_1) \qquad \mathrm{PP}^{s_1(\beta)}(\mathcal{G}, s_2)}{\mathrm{PP}^{\beta}(\mathcal{G}, s_1; s_2)}\ \text{PP-CONCAT}$$

$$\frac{\mathrm{PP}^{\beta \wedge b}(\mathcal{G}_1, s_1) \qquad \mathrm{PP}^{\beta \wedge \neg b}(\mathcal{G}_2, s_2)}{\mathrm{PP}^{\beta}(\mathcal{G}_1 \sqcup \mathcal{G}_2, \textbf{if}\ (\ b\ )\ s_1\ \textbf{else}\ s_2)}\ \text{PP-IF} \qquad \frac{\mathrm{PP}^{\beta \wedge b}(\mathcal{G}, s)}{\mathrm{PP}^{\beta}(\mathcal{G}, \textbf{while}\ (\ b\ )\ s)}\ \text{PP-WHILE}$$

Fig. 18. The PP-system

Note that $\mathcal{G}_b$ guarantees the same number of iterations in both executions; consequently, for a terminating loop (non-termination is not considered), σ_1^k and σ_2^k will fall in the first case (false guard) after the same number k of iterations. These states are exactly σ_1' and σ_2', and agree on $\mathcal{G} \sqcap \mathcal{G}_b$ after the loop. □

THEOREM 7.12 (G-SOUNDNESS). *Let s be a statement, $\mathcal{G}'$ be required after s, β be a predicate and p be the program point before s. Let also $\mathcal{G}$ be an agreement computed before s by means of the* G*-system. Let τ_1 and τ_2 be two trajectories, and the states $\sigma_1 \in \tau_1[p]$ and $\sigma_2 \in \tau_2[p]$ satisfy $\mathcal{G}(\sigma_1, \sigma_2)$ and β. Then, the condition $\mathcal{G}'(\sigma_1', \sigma_2')$ holds, where $\sigma_i' = [\![s]\!](\sigma_i)$.*

PROOF. Easy from Lemmas 7.9 and 7.11, and the discussion explaining each rule (especially, G-FASSIGN). □

7.1.7. The PP*-system.* Property preservation can be proved by means of a rule system, the PP-system (Figure 18). Most rules are straightforward or very similar to G-system rules, and characterize when executing a certain statement preserves the properties represented by an agreement $\mathcal{G}$ (i.e., for every variable x, the property/uco $\mathcal{G}(x)$ is preserved).

For example, rule PP-ASSIGN allows proving that a certain agreement is preserved when the initial value of x cannot be distinguished from the value of the expression (i.e., the new value of x), when it comes to the property $\mathcal{G}(x)$. In rule PP-FASSIGN, a mechanism similar to G-FASSIGN is used: definite aliasing and possible sharing can be used to identify which variables are affected by the field update. As for G-FASSIGN, an optimization based on field-sensitive sharing analysis (Section 7.4.2) can be introduced, which makes it easier to prove property preservation on field updates.

Indeed, a number of optimizations can be applied to the PP-system (for example, one could think that property preservation on $s_1; s_2$ does not require the preservation of the same properties on both statements separately). However, this rule system is not the central part of this paper, and Figure 18 is just a sensible way to infer property preservation.

7.2. Agreements and slicing criteria

This approach to compute abstract slices follows the standard conditioned, non-iteration-count form of backward slicing. Therefore, a slicing criterion C takes the form $(\mathcal{I}, \mathcal{X}, \{n\} \times \mathbb{N}, \mathit{false}, \mathcal{A})$ where n is the last program point, and $\mathcal{A}$ is a sequence of ucos

assigning a property to each variable in $\mathcal{X}$. The initial agreement can be easily computed from the criterion and is such that $\mathcal{G} = \{[x::\mathcal{A}_x] \mid x \in \mathcal{X}\}$, where $\mathcal{A}_x$ is the element of $\mathcal{A}$ corresponding to x. This shows that there is a close relation between this specific kind of slicing criteria and agreements, and in the following, these concepts will be used somehow interchangeably in informal parts. It will be shown that this makes sense, i.e., criteria and agreements define tightly related notions. Next definition defines the correctness of an abstract slice of this kind, where the slicing criterion is intentionally confused with an agreement.

DEFINITION 7.13 (ABSTRACT SLICING CONDITION). *Let P^s be the slice of P with respect to an agreement (criterion) $\mathcal{G}$. In order for P^s to be correct, $[\![P]\!](\sigma)$ and $[\![P^s]\!](\sigma)$ must agree on $\mathcal{G}$ for every initial σ: $\mathcal{G}([\![P]\!](\sigma), [\![P^s]\!](\sigma))$.*

7.3. Erasing statements

The main purpose of the G-system is to propagate a final agreement backwards through the program code, in order to have a specific agreement attached to each statement[12]. This is done as follows: a program can be seen as a sequence $\overline{s} = s_0; ...; s_k$ of $k+1$ statements, where each s_i can be either a simple statement (skip, assignment, field update) or a compound one (conditional or loop), containing one (the loop body) or two (the branches of the conditional) sequences of statements (either simple or compound, recursively). The way to derive an agreement for every statement in the program is depicted in the pseudocode of Figure 19. The procedure labelSequence takes as input

(1) a sequence of statements (in the first call, it is the whole program code[13]);
(2) a pair of agreements: (2.a) the first one, $\mathcal{G}_{in}$, refers to the beginning of the sequence, and, in the first call, is such that the abstraction on each variable is ρ_{ID}; and (2.b) the second one, $\mathcal{G}_{out}$, is the desired final agreement, which corresponds to the slicing criterion as discussed in Section 7.2; and
(3) a predicate on states which is supposed to hold at the beginning of the sequence.

labelSequence goes backward through the program code inferring, for each s_i, an agreement $\mathcal{G}_i$ which corresponds to the program point *after* s_i. $\mathcal{G}_k$ will be the same $\mathcal{G}_{out}$, whereas, for each i, $\mathcal{G}_{i-1}$ will be inferred by using the G-system: more specifically, it is a (ideally, the best) precondition such that the tuple $\{\mathcal{G}_{i-1}\}^{\beta_{i-1}}\ s_i\ \{\mathcal{G}_i\}$ holds. Note that, since the initial $\mathcal{G}_0$ is the identity on all variables, $\{\mathcal{G}_0\}^{\beta_0}\ s_1; ..; s_i\ \{\mathcal{G}_i\}$ trivially holds for every i (execution is deterministic); however, the $\mathcal{G}_{in}$ argument plays an important role when dealing with loop statements.

Statements inside compound statements (e.g., assignments contained in the branch of a conditional) are also labeled with agreements: this is done by calling labelSequence recursively. Note that, in this case, if $\overline{s}_t$ and $\overline{s}_f$ are, respectively, the sequences corresponding to the "then" and "else" branch of a conditional statement s_i, then labelSequence is called with second argument $(\mathcal{G}_{in}, \mathcal{G}_i)$; this is so because the state does not change when control goes from the end of a branch to the statement immediately after s_i.

The treatment of loops follows closely the definition of the rule G-WHILE. In the augmented triple at line 16, $\mathcal{G}_{i-1}$ appears before and after the statement; this is consistent with the rule. The condition $\mathcal{G}_{i-1} \sqsubseteq \mathcal{G}_i \sqcap \mathcal{G}_b$ guarantees that the $\{\mathcal{G}_{i-1}\}^{\beta_{i-1}}\ s_i\ \{\mathcal{G}_i\}$ can be proven by applying the G-SUB rule. Moreover, the recursive call on the body $\overline{s}_l$ at line 19 has

[12]Here, a statement is not only a piece of code, but also a position (program point) in the program, so that no two statements are equal, even if they are syntactically identical. For the sake of readability, the program point is left implicit.
[13]Strictly speaking, the sequence of statements is the program without the initial sequence of **read** statements: remember that **read** statements are basically meant to provide the input.

```
1  procedure labelSequence('s_1;...;s_k', (G_in, G_out), β) {
2    G_0 = G_in;
3    G_k = G_out;
4    every β_i is s̄(β) where s̄ = 's_1;...;s_i';
5    // (remember the transformed predicate s(β))
6    for i = k downto 1 {
7      if (s_i is a conditional) {
8        let b be the guard;
9        let s̄_t and s̄_f be its branches;
10       call labelSequence(s̄_t, (G_0, G_i), β_{i-1} ∧ b);
11       call labelSequence(s̄_f, (G_0, G_i), β_{i-1} ∧ ¬b);
12       G_{i-1} is such that {G_{i-1}}^{β_{i-1}} s_i {G_i}; }
13     else if (s_i is a loop) {
14       G_{i-1} is an agreement such that
15       - G_{i-1} ⊑ G_i ⊓ G_b // (see rule G-WHILE)
16       - {G_{i-1}}^{β_{i-1}} s_i {G_{i-1}}
17       let b be the guard;
18       let s̄_l be the loop body;
19       call labelSequence(s̄_l, (G_{i-1}, G_{i-1}), β_{i-1} ∧ b); }
20     else { // non-compound statement
21       G_{i-1} is such that {G_{i-1}}^{β_{i-1}} s_i {G_i}; } }
22 }
```

Fig. 19. Labeling program code with agreements by using the G-system

$(\mathcal{G}_{i-1}, \mathcal{G}_{i-1})$ as its second argument. In this recursive call, the value of the formal parameter $\mathcal{G}_{in}$ will no longer be, in general, the identity on all variables. The code fragment dealing with loops (lines 14 to 19) relies on an agreement $\mathcal{G}_{i-1}$ satisfying a number of conditions (line 15, line 16, and the success of the recursive call at line 19). It is important to note that such an agreement always exists, so that the execution of the pseudo-algorithm never stops: the $\mathcal{G}_{i-1}$ satisfying all conditions is the identity on all variables.

The following proposition specifies when it is correct to slice out a statement from a program.

PROPOSITION 7.14. *Consider a program P. Suppose that the judgment $\text{PP}^{\beta}(\mathcal{G}_s, s)$ can be proved, where $\mathcal{G}_s$ and β_s are computed by* labelSequence*, and β refers to the program point before s. In this case, let P' be the program P where s has been replaced by* **skip***, and σ be an initial state. Then, the following holds: upon termination (non-termination is not considered), $[\![P]\!](\sigma)$ agrees with $[\![P']\!](\sigma)$ with respect to the agreement $\mathcal{G}_{out}$ corresponding to the desired slicing criterion.*

PROOF. Let τ and τ' be two trajectories coming from executing, respectively, P and P' from the initial state σ. Let $\sigma_{in}[j]$ and $\sigma'_{in}[j]$ be the states of τ and τ', respectively, when control reaches the program point before s (or **skip**, in the case of P') for the j-th time, and $\sigma_{out}[j]$ and $\sigma'_{out}[j]$ be their corresponding states after s (or **skip**). If s is not contained in any loop, then j can only be 1, and the proof is trivial. Otherwise, it can be any number up to some k_s (a non-negative number).

If $k_s = 0$, then the loop is never executed on the input σ, and the proof follows trivially.

Otherwise, $\sigma_{in}[1]$ and $\sigma'_{in}[1]$ are identical because both executions have been going exactly through the same statements; as a consequence, they certainly agree on $\mathcal{G}_s$. On the other hand, $\sigma_{out}[1]$ and $\sigma'_{out}[1]$ are in general not identical, but they still agree on $\mathcal{G}_s$ by the

following:

(1) $\mathcal{G}_s(\sigma_{in}[1], \sigma'_{in}[1])$ (they are identical)
(2) $\mathcal{G}_s(\sigma'_{in}[1], \sigma'_{out}[1])$ (semantics of **skip**)
(3) $\mathcal{G}_s(\sigma_{in}[1], \sigma_{out}[1])$ (property preservation on s)
(4) $\mathcal{G}_s(\sigma_{out}[1], \sigma'_{out}[1])$ (transitivity applied to (1), (2) and (3))

Due to how the program is labeled with agreements by labelSequence, the agreement $\mathcal{G}_s(\sigma_{out}[1], \sigma'_{out}[1])$ after the first iteration implies an agreement of both executions at the end of the loop body with respect to the agreement $\mathcal{G}_{end}$ labeling that program point. This also means that both executions will still agree on $\mathcal{G}_{end}$ at the beginning of the next (second) iteration, and (again, by construction) they will also agree on $\mathcal{G}_s$ when s is reached for the second time. This mechanism can be repeated until k_s is reached, and it is easy to realize that the agreement on $\mathcal{G}_s$ at the last iteration implies the final agreement on $\mathcal{G}_{out}$ (see Figure 20 for a picture).

The crux of this reasoning is that, by construction, agreements labeling each program point imply that, when a statement can be removed from the body, P and P' will execute the loop body the same number of times. In general, this does not mean that every loop will be executed the same number of times (for example, a property-preserving loop could be even sliced out completely as in Example 7.17). □

Example 7.15. Consider the following program:

```
23 while (y>0) { // {[x::ρ_PAR], [x::ρ_SIGN]}
24   if (...) { x := x+1; }
25   ...
26   x := x−2; // {[x::ρ_PAR], [x::ρ_SIGN]}
27 } // {[x::ρ_PAR]}
```

Given the final agreement, the loop cannot be entirely sliced out because parity may change at line 24. The agreement after lines 23 and 26 is the $\mathcal{G}_{i-1}$ introduced at line 14 of labelSequence; it clearly satisfies the condition at line 15 of the same algorithm, and we suppose that it also satisfies the augmented triple of line 16. Then, to remove the statement at line 26 is irrelevant to the slicing criterion since it does not modify the parity of x nor the sign of y.

This proposition can be used in order to remove all statements for which such a property-preservation judgment can be proved. In general, the slice is computed by replacing all statements s such that $\text{PP}^{\beta'}(\mathcal{G}_s, s)$ holds by **skip**, or, equivalently, removing all of them from the original code. It is easy to observe that this corresponds exactly to the notion of backward abstract slicing given in Section 4.

Example 7.16. Consider the following code, and let the final nullity of x be the property of interest (corresponding to the agreement $\{[x::\rho_{\text{NULL}}]\}$ after line 15):

```
8  ...
9  n := n*2;          // {[n::ρ_ZERO]}
10 C x := new C();    // {[n::ρ_ZERO]}
11 if (n=0) {         // {} (final result on this branch always null)
12   x := null;       // {[x::ρ_NULL]}
13 } else {           // {} (final result on this branch never null)
14   x := new C();    // {[x::ρ_NULL]}
15 }                  // {[x::ρ_NULL]}
```

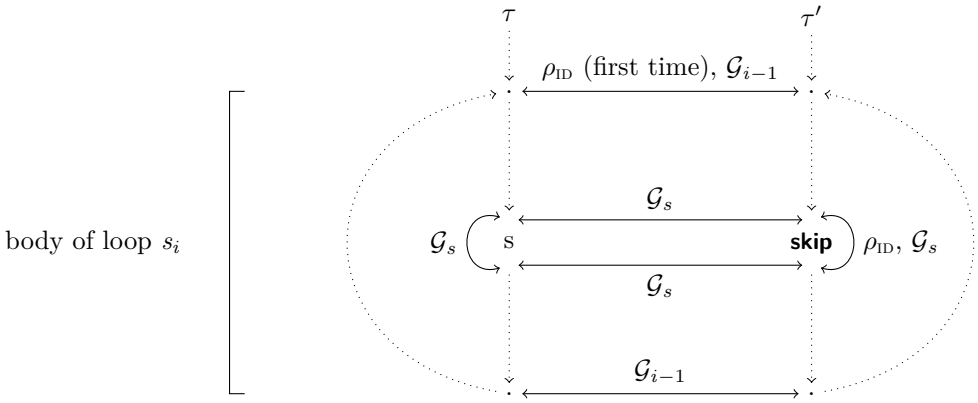


Fig. 20. Graphical representation of some aspects of Proposition 7.14. τ and τ' are the executions of P and P', respectively. Labeled horizontal double arrows indicate the (provable) agreements between both traces at different program points: (a) at the beginning of the loop body; (b) before s or **skip**; (c) after s or **skip**; and (d) at the end of the body. Moreover, both s and **skip** preserve $\mathcal{G}_s$.

Each agreement on the right-hand side is the label after the statement at the same line. Both lines 9 and 10 can be removed from the slice because:

— the final nullity of x only depends on the equality of n to 0 before line 11, which is captured by the ρ_{ZERO} domain;
— line 10 does not affect n; and
— multiplying a number by 2 preserves the property of being equal to 0.

Note that, although agreements after lines 11 and 13 are both empty (no matters how the state is, x will be null after line 12 and non-null after line 14), the one after line 10 is not because of rule G-IF2.

Example 7.17. . Consider again the code of Example 4.1. Starting from the final agreement $\{[\,\mathsf{list} ::\rho_{\text{CYC}}]\}$, the code is annotated as follows:

```
33                          // {[list::ρCYC]}
34 y := null;               // {[list::ρCYC]}
35 x := list;               // {[list::ρCYC], [x::ρCYC]}
36 while (pos>0) {          // {[list::ρCYC], [x::ρCYC]}
37   y := x;                // {[list::ρCYC], [x::ρCYC]}
38   x := x.next;           // {[list::ρCYC], [x::ρCYC]}
39   pos := pos-1;
40 }                         // {[list::ρCYC], [x::ρCYC]}
41 z := new Node(elem);     // {[list::ρCYC], [x::ρCYC]}
42 z.next := x;             // {[list::ρCYC], [z::ρCYC], [x::ρCYC]}
43 if (y = null) {          // {[z::ρCYC], [x::ρCYC]}
44   list := z;             // {[list::ρCYC]}
45 } else {                  // {[list::ρCYC], [z::ρCYC], [x::ρCYC]}
46   y.next = z;            // {[list::ρCYC]}
47 }                         // {[list::ρCYC]}
```

To prove the necessary tuples, it is important to note that next is the only reference field selector in the class Node, so that any cycle has to traverse it. Moreover, to prove that the cyclicity of list after line 46 is related to the cyclicity of list , z and x before that line needs non-trivial reasoning about data structures; concretely, it is necessary to have some

reachability analysis [Genaim and Zanardini 2013][14] capable to detect that there is no path from z to y (otherwise, a cycle could be created by y.next = z). This information should be available as β[15], and allows to say that y and list (note that list is affected because it is sharing with y) are cyclic after line 46 if and only if z or x or list were before that line (actually, a closer inspection of the code reveals that z, x and list are either all cyclic or all acyclic before line 46). Variable y does not appear in the agreement at line 45 because next is the only field, so that any cycle possibly reachable from it before line 46 is no longer reachable after that same line.

The situation is similar at line 42: the cyclicity of list and x is not affected by the field update since z is provably not sharing with any of them before line 42. On the other hand, the cyclicity of z depends on x and list because any cycle reachable from z after line 42 must be also reachable from the other two variables.

Moreover, the agreement does not change in the loop body at lines 36–40 because (1) data structures are not modified; (2) list is not affected in any way; and (3) the value of x changes, but its cyclicity does not (by executing x:=x.next, there is no way to make unreachable a cycle which was reachable before, or the other way around). Importantly, this also means that the cyclicity of list and x is *preserved* by the loop, so that it can be safely removed from the slice (the preserved property is the same as the agreement after line 40).

On the other hand, the conditional statement at lines 43–47 cannot be removed directly because it is not possible to prove that the cyclicity of list is preserved through it (actually, it is not). In order to prove that the whole code between lines 34 and 47 preserves the cyclicity of list , a kind of *case-based* reasoning could be used: (1) the initially acyclicity of list implies its final acyclicity; and (2) the initially cyclicity of list implies its final cyclicity. Both these results can be proved by standard static-analysis techniques [Genaim and Zanardini 2013].

Needless to say, slices could be sub-optimal (for example, the requirement about executing loops the same number of times needs not be satisfied by any correct slice). It is not difficult to see that, if a "concrete" slicing criterion would be considered instead of an abstract one, then agreements would only be allowed to contain conditions $[x::\rho_{\text{ID}}]$ for a certain set of variables. The labelSequence would work exactly the same way, with an important difference: when trying to compute the precondition of a tuple $\{\_\}^{\beta}\ s\ \{\mathcal{G}\}$, the possible outcome could, again, contain only conditions $[x::\rho_{\text{ID}}]$.

Example 7.18. In Example 7.16, suppose the final agreement be $\{[x::\rho_{\text{ID}}]\}$, corresponding to a concrete slicing criterion interested in (the exact value of) x. In this case, the agreement after line 10 could only be $\{[n::\rho_{\text{ID}}]\}$, since (1) $\{\}$ would not be correct, and (2) no other abstract domain can appear in agreements. This way, line 9 could not be sliced out since it does not preserve the ρ_{ID} property of n.

Semantically, abstract slices are in general smaller than concrete slices. This is also the case of Example 7.18. Clearly, this does *not* imply that *every* abstract slicing algorithm would remove more statements than *every* concrete slicing algorithm.

[14]Note that pair-sharing-based cyclicity analysis [Rossignoli and Spoto 2006] is not enough since y and z are sharing before line 46.

[15]See Section 2.4 and the beginning of this section for a discussion about β and similar examples of statically-inferred information about states.

7.4. Practical issues and optimizations

This section discusses how the analysis can realistically deal with the computation of abstract dependencies and the propagation of agreements, and an optimization based on recent work on sharing.

7.4.1. Agreements and ucos. One of the major challenges of the whole approach is how agreements are propagated backwards through the code as precisely as possible, i.e., being able to detect that agreement on some ucos before s implies agreement on (possibly) other ucos after s.

Ideally, given s, β and $\mathcal{G}'$, the G-system should find the *best* $\mathcal{G}$ such that $\{\mathcal{G}\}^{\beta}\ s\ \{\mathcal{G}'\}$. However, it is clearly unrealistic to imagine that the static analyzer will always be able to find the best ucos without going into severe scalability (even decidability) issues: in general, there exist infinite possible choices for an input agreement satisfying the augmented triple. In practice, an implementation of this slicing algorithm will be equipped with a library of ucos among which the satisfaction of augmented triples can be checked. The wider the library, the more precise the results. Rule of the PP-system and the G-system can be specialized with respect to the ucos at hand.

Example 7.19. Consider this code already presented in Example 7.16, where the slicing criterion is the final nullity of x:

```
 8  ...
 9  n := n*2;
10  C x := new C();
11  if (n=0) {
12    x := null;
13  } else {
14    x := new C();
15  }
```

In order to be able to remove lines 9 and 10 from the slice, an analyzer has to "know" that, after merging the result of both branches, the uco ρ_{ZERO} precisely describes the agreement before line 11. In other words, the analyzer must know both ρ_{ZERO} and ρ_{NULL} in order to be able to manipulate information about them. Moreover, given a library of ucos, rules can be optimized for some recurrent programming patterns like guards (n=0) or (x=**null**), or statements m:=0.

It is clear that to design of an analyzer which is able to deal with all possible ucos is infeasible. However, the combination of some simple numeric or reference domains like the one described in this paper would already lead to meaningful results.

7.4.2. Use of Field-Sensitive Sharing. As already mentioned, *field-sensitive sharing analysis* [Zanardini 2015] is able to keep track of fields which are involved in *converging paths* from two variables to a common location in the heap (the *shared* location). A *propositional formula* is attached to each pair of variables and each program point, and specifies the fields involved in *every* pair of converging paths reaching a common location. For example, if the formula $\neg\not{\mathsf{f}}_{\searrow} \wedge \not{\mathsf{g}}_{\swarrow}$ is attached to a pair of variables (x, y) at a certain program point n (written $S_n(x, y) = \neg\not{\mathsf{f}}_{\searrow} \wedge \not{\mathsf{g}}_{\swarrow}$), this means that the analysis was able to detect that, for *every* two paths π_1 and π_2 in the heap starting from x and y, respectively, and both ending in the same (shared) location,

— π_1 certainly does not traverse field f, as dictated by $\neg\not{\mathsf{f}}_{\searrow}$ (arrows from top-left to bottom-right refer to paths from x, i.e., the first variable in the pair under study); and

— π_2 certainly traverses g, as prescribed by $\swarrow g$ (arrows from top-right to bottom-left refer to paths from y).

In presence of such an analysis, two kinds of improvements can be potentially obtained when analyzing a field update:

— the number of variables which can be actually affected by an update is, in general, reduced since it is possible to guarantee that some (traditionally) sharing variables will not be affected;
— even for variables which are (still) possibly sharing with x, the set of field sequences $\bar{g}$ to be considered can be substantially smaller.

Example 7.20. Suppose that field-sensitive sharing analysis is able to guarantee the following at a program point before the field update x.f:=e:

— The formula $\neg \not\searrow \wedge \swarrow g$ correctly describes the sharing between x and y; and
— The formula $\swarrow h$ correctly describes the sharing between x and z.

According to the traditional notion of sharing, both y and z may share with x. However,

— the assignment x.f:=e provably does *not* affect y because no path from x traversing f will reach a location that is also reachable from y; and
— when considering all the possible field sequences starting from z, only those containing h have to be considered.

The rule G-FASSIGN can be refined by using field-sensitive sharing, as follows. Let n be the program point before the field update.

— in pre-condition $(**)$, the only sharing variables that have to be dealt with are those for which it cannot be proved that they are unaffected by the update; this can be done by defining a new set $\text{SH}_f(x)$ of variables which are possibly sharing with x in such a way that some path from x to a shared location could traverse f:

$$\text{SH}_f(x) = \{y \mid S_n(x,y) \not\models \neg \not\searrow\}$$

This means that y is considered as potentially affected by the update when the propositional formula describing how it shares with x does not entail that paths from x to shared locations do not traverse f.
— in the same pre-condition $(**)$, the universal quantification on field sequences can be restricted to those compatible with field-sensitive information. More formally, a field sequence $\langle .f_1.f_2.....f_n\rangle$ has to be considered only if it is possible that a path from y traversing exactly those fields ends in a shared location, or, equivalently, if the set $\{\not\searrow, \swarrow f_1, \swarrow f_2, .., \swarrow f_n\}$ is a *model* of $S_n(x,y)$. That such a set is a model of $S_n(x,y)$ is equivalent to say that the field-sensitive information is compatible with the existence of a pair of paths π_1 and π_2 such that (1) π_1 starts from x; (2) π_2 starts from y; (3) π_1 only traverses f; (4) π_2 traverses all and only the fields $f_1, f_2, ...f_n$; and (5) both paths end in the same shared location. Let $\mathit{fields}(\bar{g})$ be the set of fields contained in the field sequence $\bar{g}$. Then, the above condition can be written as

$$\text{SH}^{x,y}_{seq}(f,\bar{g}) \quad \equiv \quad \bigwedge_{p\in X} p \wedge \bigwedge_{q\notin X} \neg q \quad \models \quad S_n(x,y)$$

where $X = \{\not\searrow\} \cup \{\swarrow g \mid g \in \mathit{fields}(\bar{g})\}$, and $q \notin X$ means that q is any proposition $\not\searrow h$ or $\swarrow h$ (for some field h) not included in X.

The refined G-FASSIGN rule, called G-FASSIGN2, comes to be

$$\frac{\begin{array}{ll}(*) & \forall y \in \mathrm{DAL}(x)\,.\,\forall \sigma_1 \models \beta, \sigma_2 \models \beta.\\ & \quad \mathcal{G}(\sigma_1,\sigma_2) \Rightarrow\\ & \quad \mathcal{G}'(\sigma_1[y.f \leftarrow [\![e]\!](\sigma_1)], \sigma_2[y.f \leftarrow [\![e]\!](\sigma_2)])\\ (**) & \forall y \in \mathrm{SH}_f(x)\,.\,\forall \bar{g}.\ \mathrm{SH}^{x,y}_{seq}(f,\bar{g}) \Rightarrow (\forall \sigma_1 \models \beta, \sigma_2 \models \beta.\\ & \quad \mathcal{G}(\sigma_1,\sigma_2) \Rightarrow\\ & \quad \mathcal{G}'(\sigma_1[y.\bar{g} \leftarrow [\![e]\!](\sigma_1)], \sigma_2[y.\bar{g} \leftarrow [\![e]\!](\sigma_2)]))\\ (***) & \forall y \notin \mathrm{DAL}(x)\,.\,\mathcal{G}(y) \sqsubseteq \mathcal{G}'(y)\end{array}}{\{\mathcal{G}\}^{\beta}\ \ x.f{:=}e\ \ \{\mathcal{G}'\}}\ \textsc{g-fassign}2$$

7.5. Comparison with related algorithms

The Tukra Abstract Program Slicing Tool [Halder and Cortesi 2012] implements the computation of a Dependence Condition Graph [Halder and Cortesi 2013] for performing abstract slicing. To use a Program Dependence Graph is somehow alternative to the computation of agreements. As far as the author make it possible to understand, Tukra only deals with numerical values, and it is not clear which properties are supported (i.e., which is the "library" of ucos mentioned in Section 7.4.1).

Moreover, the authors of that tool point out that their approach is able to exclude some dependencies that were not ruled out in previous work introducing abstract dependencies [Mastroeni and Zanardini 2008]. However, they do not consider that a rule system for computing agreements (essentially, the G-system described in the present paper) was introduced [Zanardini 2008] before Tukra was developed, and does not suffer from the limitations they describe.

8. RELATED WORK

The formal framework referred to in this paper [Binkley et al. 2006a] is not the only attempt to provide a unified mathematical framework from program slicing. In [Ward and Zedan 2007], the authors have precisely this aim. In this work, the authors unify different approaches to program slicing by defining a particular semantic relation, based on the weakest precondition semantics, called *semirefinement* such that, given a program P, the possible slices are all the programs that are semirefinements of P. In this framework, different forms of slicing are modeled as program transformations. Hence, a program Q is a slice of P if the transformation of Q (corresponding to the particular form of slicing to compute) is a semirefinement of the same transformation of P. This approach is extremely interesting, but does not really allow to compare the different forms of slicing, feature that we consider fundamental for introducing the new abstract forms of slicing as generalizations of the existing ones. It may surely deserve further research to study whether also abstract slicing could be modeled in this framework.

As far as the relation between slicing and dependencies is concerned, there are at least two works that are related with our ideas in different ways. One of the first works aiming at formalizing a semantic approach to dependency, leading to a semantic computation of slicing, is the *information-flow logic* by Amtoft and Banerjee [Amtoft and Banerjee 2007]. This logic allows us to formally derive, by structural induction, the set of all the *independencies* among variables. In Figure 21, the original notation proposed by the authors is used, where $[x \ltimes y]$ is to be 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 on which at least one test surrounding the statements depends on.

In our aim of defining slicing in terms of dependencies, the first thing we have to observe in this logic is that it always computes (in)dependencies from the *initial* val-

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

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

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

Fig. 21. A fragment of the independency logic

ues of variables. This makes its use for slicing not so straightforward, since it loses the *local* dependency between statements. 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 logic in Fig. 21, we lose the trace of (in)dependencies which, in this case, would involve all the three assignments. As a matter of fact, this logic is more suitable for forward slicing, which is the one considered by the authors [Amtoft and Banerjee 2007], since it fixes the criterion *on the input*. 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 statements [Amtoft and Banerjee 2007]. In the logic, more explicitly, this notion of dependency is used for characterizing the set of independencies holding during the execution of a program.

Another, more recent, approach to slicing by means of dependencies is [Danicic et al. 2011]. In this work, the authors propose new definitions of control dependencies: non-termination sensitive and insensitive. These new semantic notions of dependencies are then used for computing more precise standard slices. It could be surely interesting to study the semantic relation between their notion of dependencies and the ones we propose in this paper.

Finally, a related algorithm for computing abstract slices has been already discussed in Section 7.5. It is necessary to point out that the agreement-based approach to abstract slicing [Zanardini 2008] was introduced before the Tukra tool.

9. CONCLUSION AND FUTURE WORK

The present paper formally defines the notion of abstract program slicing, a general form of slicing where properties of data are observed instead of their exact value. A formal framework is introduced where the different forms of abstract slicing can be compared; moreover, traditional, non-abstract forms of slicing are also included in the framework, allowing to prove that non-abstract slicing is a special case of abstract slicing where no abstraction on data is performed.

Algorithms for computing abstract dependencies and program slices are given. Future work includes an implementation of this analysis for an Object-Oriented programming language where properties may refer either to numerical or reference values (to the best of our knowledge, existing tools only deal with integer variables). On the other hand, we observed that the provided notion of abstract dependency is not suitable for slicing computation by using PDGs. We believe that it is possible to further generalize the notion of abstract dependencies allowing to characterize a recursive algorithm able to track backwards both the

variables that affect the criterion, and the abstract properties of these variables affecting the abstract criterion.

Another interesting line of research is to understand how other approaches to slicing can be extended in order to include abstract slicing. As noted before, it would be interesting to study whether it is possible to model abstract slicing as a program transformation, allowing us to define also abstract slicing in term of semirefinement [Ward and Zedan 2007]. Another, more algorithmic, interesting approach is the one proposed in [Barros et al. 2010], where weakest precondition and strongest postcondition semantics are combined in a new more precise algorithm for standard slicing. It could be very interesting to understand whether this approach could be extended in order to cope also with the computation of abstract forms of slicing.

ACKNOWLEDGMENTS

This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

REFERENCES

T. Amtoft and A. Banerjee. 2007. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming* 64, 1 (2007), 3–28.

Jose Bernardo Barros, Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. 2010. Assertion-based Slicing and Slice Graphs. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM '10)*. IEEE Computer Society, Washington, DC, USA, 93–102.

A. Bijlsma and R.P. Nederpelt. 1998. Dijkstra-Scholten predicate calculus : concepts and misconceptions. *Acta Informatica* 35, 12 (1998), 1007–1036.

D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. 2006a. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming* 62, 3 (2006), 228–252.

D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. 2006b. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science* 360, 1 (2006), 23–41.

D. W. Binkley and K. B. Gallagher. 1996. Program Slicing. *Advances in Computers* 43 (1996).

G. Canfora, A. Cinitile, and A. De Lucia. 1998. Conditioned program slicing. *Information and Software Technology* 40 (1998), 11–12.

Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. 1996. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance* 8, 3 (1996), 145–178.

P. Cousot. 2001. Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics - 10 Years Back. 10 Years Ahead.* 138–156.

P. Cousot and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 238–252.

P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, 269–282.

Sebastian Danicic, Richard W. Barraclough, Mark Harman, John D. Howroyd, Ákos Kiss, and Michael R. Laurence. 2011. A Unifying Theory of Control Dependence and Its Application to Arbitrary Program Structures. *Theor. Comput. Sci.* 412, 49 (2011), 6809–6842.

A. De Lucia. 2001. Program Slicing: Methods and Applications. In *Proceedings of International Workshop on Source Code Analysis and Manipulation (SCAM)*.

E. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

E. Dijkstra and C. S. Scholten. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag.

John Field, G. Ramalingam, and Frank Tip. 1995. Parametric program slicing. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 379–392.

K. B. Gallagher and J. R. Lyle. 1991. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* 17, 8 (1991), 751–761.

Samir Genaim and Damiano Zanardini. 2013. Reachability-based Acyclicity Analysis by Abstract Interpretation. *Theoretical Computer Science* 474, 0 (February 2013), 60–79.

Roberto Giacobazzi, Neil D. Jones, and Isabella Mastroeni. 2012. Obfuscation by partial evaluation of distorted interpreters. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*. 63–72.

R. Giacobazzi and I. Mastroeni. 2004a. Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 186–197.

R. Giacobazzi and I. Mastroeni. 2004b. Proving Abstract Non-Interference. In *Annual Conf. of the European Association for Computer Science Logic (CSL '04)*, A. Tarlecki J. Marcinkowski (Ed.), Vol. 3210. Springer-Verlag, Berlin, 280–294.

R. Giacobazzi, F. Ranzato, and F. Scozzari. 2000. Making abstract interpretations complete. *J. of the ACM.* 47, 2 (2000), 361–416.

Raju Halder and Agostino Cortesi. 2012. Tukra: An Abstract Program Slicing Tool. In *Proceedings of International Conference on Software Paradigm Trends (ICSOFT)*. 178–183.

Raju Halder and Agostino Cortesi. 2013. Abstract program slicing on dependence condition graphs. *Science of Computer Programming* 78, 9 (2013), 1240–1263.

Michael Hind. 2001. Pointer Analysis: Haven'T We Solved This Problem Yet?. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM Press, New York, 54–61.

C.A.R Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

S. Horwitz, J. Prins, and T. Reps. 1989. Integrating non-interfering versions of programs. *ACM Transaction on Programming Languages and Systems* 11, 3 (1989).

S. Hunt and I. Mastroeni. 2005. The PER model of Abstract Non-Interference. In *Proceedings of Static Analysis Symposium (SAS) (Lecture Notes in Computer Science)*, Vol. 3672. Springer-Verlag, 171–185.

B. Korel and J. Laski. 1988. Dynamic Program Slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–183.

A. Majumdar, S. J. Drape, and C. D. Thomborson. 2007. Slicing obfuscations: design, correctness, and evaluation. In *Proceedings of ACM Workshop on Digital Rights Management (DRM)*. ACM, New York, NY, USA, 70–81.

Isabella Mastroeni. 2013. Abstract interpretation-based approaches to Security - A Survey on Abstract Non-Interference and its Challenging Applications. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.* 41–65.

I. Mastroeni and Đ. Nikolić. 2010. Abstract Program Slicing: From Theory towards an Implementation. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM) (Lecture Notes in Computer Science)*, Vol. 6447. Springer-Verlag, 452–467.

I. Mastroeni and D. Zanardini. 2008. Data dependencies and program slicing: From syntax to abstract semantics. In *Proceedings of Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. 125–134.

F. Ranzato and F. Tapparo. 2002. Making abstract model checking strongly preserving. In *Proceedings of Static Analysis Symposium (SAS) (Lecture Notes in Computer Science)*, Vol. 2477. Springer-Verlag, 411–427.

T. Reps. 1991. Algebraic properties of program integration. *Science of Computer Programming* 17 (1991), 139–215.

T. Reps and W. Yang. 1989. The semantics of program slicing and program integration. In *Proc. of the Colloq. on Current Issues in Programming Languages (Lecture Notes in Computer Science)*, J. Diaz and F. Orejas (Eds.), Vol. 352. Springer-Verlag, Berlin, 360–374.

Stefano Rossignoli and Fausto Spoto. 2006. Detecting Non-cyclicity by Abstract Compilation into Boolean Functions. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, Vol. 3855. Springer-Verlag, 95–110.

S. Secci and F. Spoto. 2005. Pair-Sharing Analysis of Object-Oriented Programs. In *Proceedings of the Interanational Symposium on Static Analysis (SAS)*. Springer-Verlag, 320–335.

F. Tip. 1995. A survey of program slicing techniques. *Journal of Programming Languages* 3 (1995), 121–181.

M. Ward and H. Zedan. 2007. Slicing as a program transformation. *ACM Transactions on Programming Languages and Systems* 29, 2 (2007).

M. Weiser. 1984. Program slicing. *IEEE Trans. on Software Engineering* 10, 4 (1984), 352–357.

D. Zanardini. 2008. The Semantics of Abstract Program Slicing. In *Proceeding of Working Conference on Source Code Analysis and Manipualtion (SCAM)*.

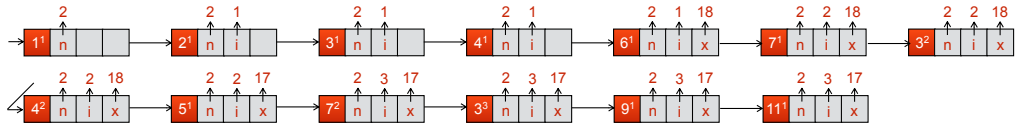
Damiano Zanardini. 2015. Field-Sensitive Sharing. *CoRR* abs/1306.6526 (2015). http://arxiv.org/abs/1306.6526

A. THE FORMAL FRAMEWORK OF PROGRAM SLICING

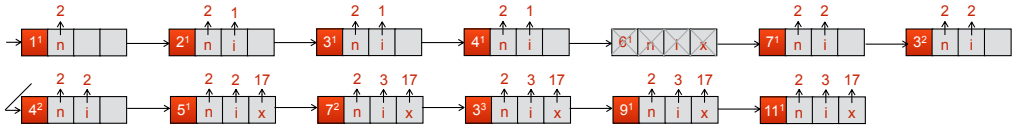
In this section, we first provide a better intuition of the differences between the forms of slicing introduced in Section 3.1 by means of examples and then we recall the main notions introduced in [Binkley et al. 2006a; 2006b] that has been generalized in this paper in the abstract form.

Different forms of slicing: some examples

Example A.1. Consider the program on the left in Figure 22. Suppose that the execution start with an initial value 2 for n (written $\mathtt{n} \leftarrow 2$). States are denoted as (m^k, μ), where m is the program point of the executed statement, k is its current iteration (i.e., the statement at m is being executed for the k-th time in the loop unrolling), and μ is the actual memory, represented by a list of pairings $x \leftarrow v$). In the picture, m^k is depicted in the first (fully colored) box, while the memory is depicted in the remaining boxes, one for each variable. A program state that is not executed in a trace is depicted by overwriting a grey cross on each box (program point and variables). The execution trajectory is the following:



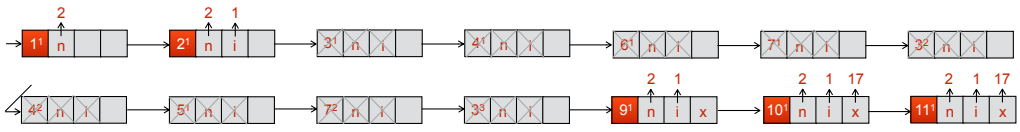
Consider now the code in the center, whose execution trajectory is the following:



precisely the same path on the statements which are in both programs (the only difference is the execution, in the original program of the statement at point 6, erased in the "candidate" slice); hence, it is a slice according both to the standard and the KL form, namely w.r.t. $C = (\{\mathtt{n} \leftarrow 2\}, \{\mathtt{x}\}, \{\langle 11, \mathbb{N}\rangle\}, \psi)$ for both possible values of ψ.

Suppose now we are interested in an IC form of slice, and the value of x at the second iteration of the program point 3. In this case, the program on the right is not a dynamic slice, since the value of x in the original program is 18, while it is undefined in the candidate slice. In other words, this program is not a slice of the program on the left w.r.t. the criterion $C = (\{\mathtt{n} \leftarrow 2\}, \{\mathtt{x}\}, \{\langle 3, \{2\}\rangle\}, \psi)$ (ψ may be both true or false).

Finally, let us consider the execution of the program on the right in Figure 22:



This last program is a standard dynamic slice since the final value of the variable of interest x is the same, but it is not a slice in the KL form, since in this last program the statement

```
1 read(n);
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }
6   else { x := 18; }
7   i := i + 1;
8 }
9 if (i = 1) {
10   x := 17; }
11 write(i,n,x);
```

```
1 read(n);
2 i := 1;
3 while (i <= n) {
4   if (i mod 2 = 0) {
5     x := 17; }
6
7   i := i + 1;
8 }
9 if (i = 1) {
10   x := 17; }
11 write(i,n,x);
```

```
1 read(n);
2 i := 1;
3
4
5
6
7
8
9 if (i = 1) {
10   x := 17; }
11 write(i,n,x);
```

Fig. 22. Programs of Example A.1

```
1 read(n);
2   i := 1;
3   i := 2;
4   if (i mod 2 = 0) {
5     x := i + n;   }
6   if (i mod 2 = 1) {
7     x := i + n + 1; }
8 write(i,n,x);
```

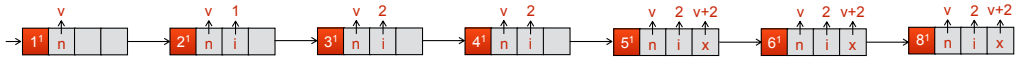
```
1 read(n);
2   i := 1;
3
4
5
6   if (i mod 2 = 1) {
7     x := i + n + 1; }
8 write(i,n,x);
```

Fig. 23. Programs of Example A.2

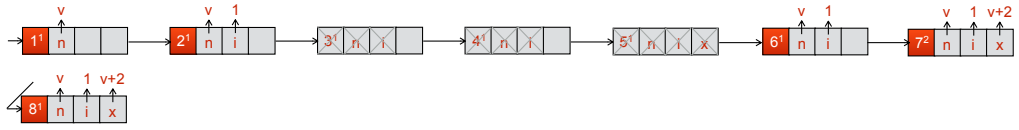
at program point 10 is executed, while in the original program it is not executed. Namely, it is a slice w.r.t. the criterion $C = (\{\mathsf{n} \leftarrow 2\}, \{\mathsf{x}\}, \{\langle 11, \mathbb{N}\rangle\}, \psi)$ only for $\psi = \mathit{false}$.

The following example shows the difference between standard and KL forms of *static* slicing.

Example A.2. Consider the program P on the left of Figure 23. Suppose static slicing is considered, i.e., all the possible initial memories are taken into account. Given an input v, the state trajectory is:



Consider now the code on the right: its state trajectory is



Consider the standard form of static slicing interested in x at program point 8^1, i.e., $C = (\{\mathsf{n} \leftarrow \mathbb{N}\}, \{\mathsf{x}\}, \{\langle 8, \{1\}\rangle\}, \mathit{false})$. Then, the program on the right is a slice of P w.r.t. C, since in 8^1 the value of x is $v + 2$ in both cases. On the other hand, if we consider the KL form, $C = (\{\mathsf{n} \leftarrow \mathbb{N}\}, \{\mathsf{x}\}, \{\langle 8, \{1\}\rangle\}, \mathit{true})$, then the program is not anymore a slice of P since there is a program point, 7^1, which is not reached in P.

The unified equivalence

The first step for defining the formal framework is to define an equivalence relation between programs, determining when a program is a slice of another. First, a *restricted memory* is obtained from a memory by restricting its domain to a set of variables. More formally, the restriction of μ with respect to a set of variables $\mathcal{X}$ is defined as $\mu \restriction \mathcal{X}$ such that $(\mu \restriction \mathcal{X})(x)$ is equal to $\mu(x)$ if $x \in \mathcal{X}$, and undefined otherwise. This restriction is used to project the

trace semantics only on those points of interest where we have to check the correspondence between the original program and the candidate slice.

The *trajectory projection* operator modifies a state trajectory by removing all those states which do not contain occurrences of program points which are relevant for the slicing criterion.

DEFINITION A.3 (TRAJECTORY PROJECTION [BINKLEY ET AL. 2006A]). *Let* $C = (\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi)$*, and* $\mathcal{L} \subseteq \mathbb{L}$ *such that* $\mathcal{L} \neq \emptyset$ *if* $\psi = true$*,* $\mathcal{L} = \emptyset$ *if* $\psi = false$*. For any* $n \in \mathbb{L}$*,* $k \in \mathbb{N}$*,* $\mu \in \mathbb{M}$*, we define the function* $Proj_0$ *as:*

$$Proj^0_{(\mathcal{X},\mathcal{O},\mathcal{L})}(n^k, \mu) \stackrel{def}{=} \begin{cases} \langle n^k, \mu \restriction \mathcal{X} \rangle & \text{if } (\exists \langle n, K \rangle \in \mathcal{O}.\ k \in K \\ \langle n^k, \bot \rangle & \text{if } (\nexists \langle n, K \rangle \in \mathcal{O}.\ k \in K) \text{ and } n \in \mathcal{L} \\ \varepsilon & \text{otherwise.} \end{cases}$$

where ε *is the empty sequence. The trace projection Proj is the extension of* $Proj^0$ *to sequences* ($\circ$ *is sequence concatenation):*

$$Proj_{(\mathcal{X},\mathcal{O},\mathcal{L})}(\langle (n_1^{k_1}, \mu_1), \ldots, (n_l^{k_l}, \mu_l) \rangle) = Proj^0_{(\mathcal{X},\mathcal{O},\mathcal{L})}(n_1^{k_1}, \mu_1) \circ \ldots \circ Proj^0_{(\mathcal{X},\mathcal{O},\mathcal{L})}(n_l^{k_l}, \mu_l)$$

$Proj^0$ takes a state from a state trajectory, and returns either one pair or an empty sequence ε. If n^k is an occurrence of interest, then it returns $\langle (n^k, \mu \restriction \mathcal{X}) \rangle$. This means that, at n, we consider exact values of variables in $\mathcal{X}$. If n^k is not an occurrence of interest, but, due to a KL form, the projection has to keep trace of a set $\mathcal{L}$ of executed statements (even if the variables in that point are not of interest), then $Proj^0$ returns $\langle (n^k, \mu \restriction \emptyset) \rangle$, meaning that we require the execution of n, but we are not interested in the values of variables in $\mathcal{X}$.

Trajectory projection allows us to define all the semantic equivalence relations characterizing on what a program and its slices have to agree due to the chosen criterion.. Given two programs P and Q, we can say that Q is a slice of P if it contains a subset of the original statements and Q is *equivalent* to P with respect to the semantic equivalence relation induced by chosen the slicing criterion.

DEFINITION A.4 ([BINKLEY ET AL. 2006A]). *Let* P *and* Q *be executable programs, and* $C = (\mathcal{I}, \mathcal{X}, \mathcal{O}, \psi)$ *be a slicing criterion. Let* $\mathbb{L}_P \subseteq \mathbb{L}$ *be the set of program points of* P*, and* $\mathcal{L} = \mathbb{L}_P \cap \mathbb{L}_Q$ *if* $\psi = true$[16] ($\mathcal{L} = \emptyset$ *if* $\psi = false$*). Then* P *is* equivalent *to* Q *w.r.t.* C *if and only if*

$$\forall \mu \in \mathcal{I}.\quad Proj_{(\mathcal{X},\mathcal{O},\mathcal{L})}(\tau_P^\mu) = Proj_{(\mathcal{X},\mathcal{O},\mathcal{L})}(\tau_Q^\mu)$$

The function $\mathcal{E}$ *maps any criterion* C *to the r to the corresponding semantic equivalence relation, hence, in this case, we write* $\langle P, Q \rangle \in \mathcal{E}(C)$*.*

Example A.5. Consider the Program P in the left of Figure 7; let the input for n be 2. Suppose we want to compute a non-iteration-count KL form of dynamic slicing, i.e., $C = (\{\mathsf{n} \leftarrow \{2\}\}, \{\mathsf{i}, \mathsf{s}\}, \langle 8, \mathbb{N} \rangle, true)$. Namely, the variables of interest are i and s, which are observed at the program point 8 each time it is reached, and the slice has not to execute statements not executed in the original program. The program on the right of Figure 7 is a slice w.r.t. C. In Figure 24 we have the execution trajectory of the original program (on the top), the execution trajectory of the candidate slice (in the middle) and the (same) projection of the two trajectories due to the chosen criterion (on the bottom).

The formal framework proposed in [Binkley et al. 2006a; 2006b] represents different forms of slicing by means of a $(\sqsubseteq, \mathcal{E})$ pair: a *syntactic preorder*, and a *function from slicing criteria*

[16]Note that, when $\mathbb{L}_Q \subseteq \mathbb{L}_P$, as we suppose in this paper, then $\mathcal{L} = \mathbb{L}_Q$. We provide the general definition since the original definition of dynamic slicing [Korel and Laski 1988] does not require that all the line of Q are included in $\mathcal{L}$; however, our choice follows the paths taken in the original framework [Binkley et al. 2006b].

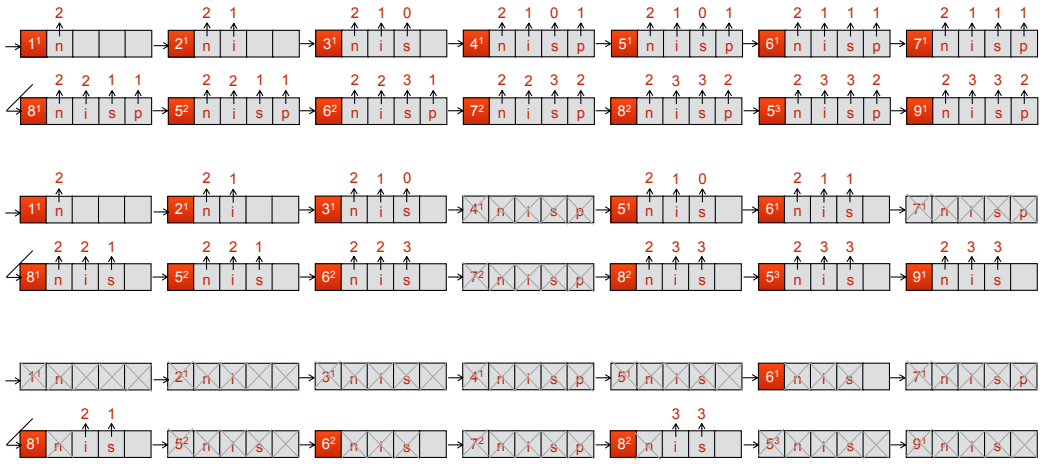


Fig. 24. Execution trace of P in Example A.5 and of the static slice and of their projection.

to semantic equivalences. The *preorder* fixes a syntactic relation between the program and its slices. In traditional slicing, slices are obtained from the original program by removing zero or more statements. This preorder is called *traditional syntactic ordering*, simply denoted by $\sqsubseteq$, and it is defined as follows: $Q \sqsubseteq P \Leftrightarrow \mathbb{L}_Q \subseteq \mathbb{L}_P$ The second component $\mathcal{E}$ fixes the semantic constraints that a subprogram has to respect in order to be a slice of the original program. As we have seen before, the equivalence relation is uniquely determined by the chosen slicing criterion determining also a specific form of slicing. This way, Binkley *et al.* are able to characterize eight forms of non-`SIM` slicing, and twelve forms of `SIM` slicing.

Finally, this framework is used to formally compare the different notions of slicing. First of all, it is defined a binary relation on slicing criteria $\rightarrow$ [Binkley et al. 2006a]: Let $C^1 = \langle \mathcal{I}^1, \mathcal{X}^1, \mathcal{O}^1, \psi^1 \rangle$ and $C^2 = \langle \mathcal{I}^2, \mathcal{X}^2, \mathcal{O}^2, \psi^2 \rangle$

$$C^1 \rightarrow C^2 \qquad \text{iff} \qquad \mathcal{I}^1 \subseteq \mathcal{I}^2,\ \mathcal{X}^1 \subseteq \mathcal{X}^2, \mathcal{O}^1 \subseteq \mathcal{O}^2, \psi^1 \Rightarrow \psi^2 \tag{2}$$

At this point, we say that a form of slicing $(\sqsubseteq, \mathcal{E}^1)$ is *weaker than* $(\sqsubseteq, \mathcal{E}^2)$ w.r.t. $\rightarrow$ iff $\forall C^1, C^2$, slicing criteria such that $C^1 \rightarrow C^2$, and $\forall P, Q$, if Q is a slice of P w.r.t. $(\sqsubseteq, \mathcal{E}^2(C^2))$, then Q is a slice of P w.r.t. $(\sqsubseteq, \mathcal{E}^1(C^1))$ as well. In this case we say that $(\sqsubseteq, \mathcal{E}^1)$ subsumes $(\sqsubseteq, \mathcal{E}^2)$. Following this definition, Binkley et al. show that all forms of slicing introduced in [Binkley et al. 2006a] are comparable in the way shown in Figure 25, where the symbols $\mathcal{S}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{SS}$, $\mathcal{SC}$ and $\mathcal{SD}$ represent static, conditioned, dynamic, static `SIM`, conditioned `SIM` and dynamic `SIM` types of slicing, respectively. Subscripts i, `KL` and `KL`i represent `IC`, `KL` and `KL`i forms of slicing, respectively; the absence of subscripts denotes the standard forms of slicing. In Figure 25 we explicitly provide both the hierarchy concerning `SIM` and non-`SIM` forms of conditioned slicing constructed in [Binkley et al. 2006a].

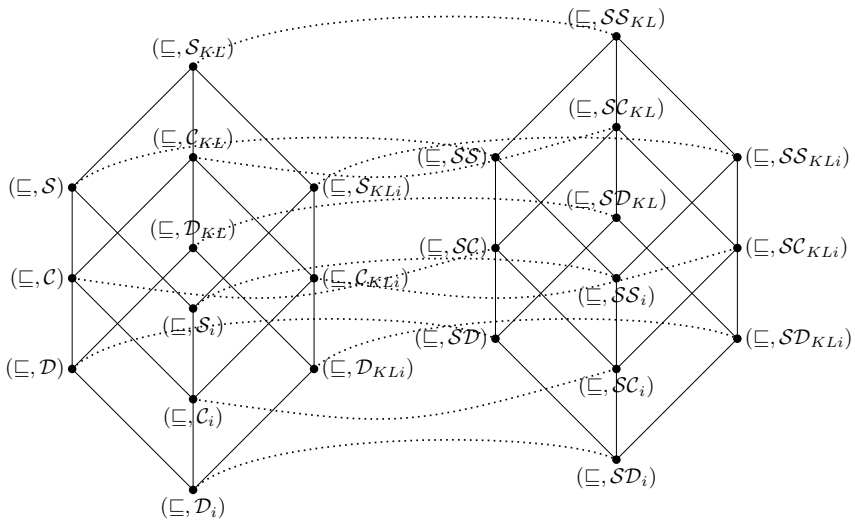


Fig. 25. Given two forms A and B, both (non-)SIM., A is weaker than B if A is connected to B by a solid line and it is below B. If A is non-SIM. and B is SIM., then A is weaker than B if A is connected to B by a dotted line and it is to the left of B.