The Semantics of Abstract Program Slicing

Damiano Zanardini
CLIP, Technical University of Madrid
E-28660 Boadilla del Monte, Madrid, Spain
damiano@clip.dia.fi.upm.es

Abstract

The present paper introduces the semantic basis for abstract slicing. *This notion is more general than standard,* concrete *slicing, in that slicing criteria are* abstract*, i.e., defined on* properties *of data, rather than concrete values. This approach is based on* abstract interpretation*: properties are* abstractions *of data. Many properties can be investigated; e.g., the nullity of a program variable. Standard slicing is a special case, where properties are exactly the concrete values. As a practical outcome, abstract slices are likely to be smaller than standard ones, since commands which are relevant at the concrete level can be removed if only some abstract property is supposed to be preserved. This can make* debugging *and* program understanding *tasks easier, since a smaller portion of code must be inspected when searching for undesired behavior. The framework also includes the possibility to restrict the input states of the program, in the style of* conditioned *slicing, thus lying between* static *and* dynamic *slicing.*

1 Introduction

The purpose of *program slicing* [27, 26, 2] is to find the part of a program (the *slice*) which is relevant to a subset of the program behavior: typically, the value of some variables at a given program point. This is specified by a *slicing criterion*, expressed as a program point and a set of variables. A slice $\mathcal{P}^s$ of $\mathcal{P}$ w.r.t. a slicing criterion S has to (i) consist of a subset of the commands of $\mathcal{P}$; (ii) be syntactically correct and executable; and (iii) give the same result as $\mathcal{P}$ if observations are restricted to S. A slice is usually computed by analyzing how the effects of a computation are propagated through the code, i.e., by inferring *dependencies*. A command must be included in the slice if it can affect the observation described by the criterion. Slicing has been widely used as an effective tool in *debugging*, *program integration*, *software maintenance* and *reverse engineering* for identifying the part of a program which is responsible for some behavior.

Main contribution The present work introduces *abstract program slicing*, a general notion based on the observation that, in typical debugging tasks, the interest is often on the part of the program which is relevant to some *property* of data, rather than their exact value. Observing properties is something which is *less precise* than observing values; e.g., if the focus is on the *nullness* of a pointer, then the corresponding observation does not need to distinguish between different non-null values.

Following the theory of *abstract interpretation* [7, 6], properties are *abstractions* of data. Consequently, slicing w.r.t. a property amounts to (1) having an *abstract slicing criterion*, where abstraction is meant to describe the property; and (2) studying dependence only as regards the abstraction, differently from the standard, *concrete* approach.

On the practical side, abstract slicing is interesting since, in general, the abstract slice on a property of some variables is *smaller* than the concrete one on the exact value of the same variables, since some code might affect the values but not the property. This can make *debugging* and *program understanding* tasks easier, since a smaller portion of the code has to be inspected when searching for some undesired behavior.

While concrete slicing algorithms are typically syntax-based, the abstract approach must rely on semantics [20]. In fact, the more abstract the property, the greater the loss of precision of the syntactic approach w.r.t. the actual semantic. The proposed technique, which is proved to be *sound*, can guide the development of abstract slicing tools, where *static analysis* will be needed to deal with semantics.

A number of variants of slicing have been proposed [26]. This paper focuses on the *backward* version, where dependencies are propagated backwards from a given program point. Related work discusses the relation of abstract slicing with *conditioned*, *static* and *dynamic* slicing. The language is *imperative* with *functions* and *structured data*.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.19

Related work Since slicing is closely related to the calculus of *dependencies* [5], which, in turn, represents one of the basic notions in *information flow* [24, 1], closely related work can be found in the *abstract non-interference* (ANI) [11] theory. There, the notion of non-interference [13] is relaxed, in the sense that flows are only detected when they affect a property (the one which can be seen by an attacker, whose observational power is limited), rather than the concrete value of data. Due to this, a program is more likely to satisfy ANI than standard non-interference, since some concrete flows are not really harmful at the abstract level. This observation is analogous to saying that abstract slices are smaller: properties *propagate less* than concrete values. More related work [12] on ANI is compared with the present approach in Sec. 4.2.

Mastroeni and Zanardini [20] discuss the notion of abstract slicing w.r.t. the *program dependency graph* approach [17], underlining that dealing with properties instead of concrete values implies *pruning* the graph and, consequently, obtaining smaller slices. This work also discusses the relation between the syntactic and semantic approaches, and provides ideas for computing *abstract dependencies*.

Rival [22] recently characterized abstract dependencies, representing data properties by means of abstract interpretation. The author discusses abstract dependence and its applications to *alarm diagnosis*, together with techniques for analyzing and composing dependencies. At a first glance, his notion of *semantic slicing* [23] seems to be similar to the present approach. However, it is quite unrelated, since it is based on *trace partitioning*, so that the slicing is on sets of traces, rather than sets of commands.

The use of *predicates* on states recalls related work on *conditioned program slicing* [4, 9, 8], a version of slicing which lies between *static* (every possible input considered) and *dynamic* [19] (only one input considered) slicing, where it is possible to consider a subset of the input states. This is done by specifying a logical formula on the program input, which identifies the set of states satisfying it (e.g., $x{\geq}0$ identifies the set of states where x is non-negative). Although this feature is not really the main focus of the present paper (the abstract slicing framework has been originally developed with static slicing in mind), conditioned slicing can be definitely useful to implement the use of predicates on states by using *symbolic execution* [18] and dependency graphs. In one sense, the semantic view of abstract slicing includes conditioned slicing as a way to specify predicates on states, but mostly uses such predicates as a way to track which path has been taken during the computation, thus increasing the precision of the analysis.

Finally, despite its title, the work by Hong, Lee and Sokolsky [16] also discusses an unrelated notion of abstract slicing. That work uses *predicates* to answer the question *for every program point, under which variable values does the program point affect the slicing criterion?* [16], and *constraints* to answer *for every program point, does the program point affect the slicing criterion if we are only interested in certain executions of a program rather than all possible ones?* [16]. As an example, consider the fragment

(**1**) $x := x + 2y + 1$;
(**2**) *if* $(x \; mod \; 2 = 1)$ (**3**) $x := x + 1$ *else* (**4**) $x := x - 1$

with the final value of x as the criterion. In this case, that notion of abstract slicing gives predicates

$$(\mathbf{1}) : true \quad (\mathbf{2}, \mathbf{3}) : x \; mod \; 2 = 1 \quad (\mathbf{4}) : x \; mod \; 2 = 0$$

meaning that commands **(2)** and **(3)** are relevant only if x is odd, while **(4)** is relevant in the opposite case and **(1)** is always relevant. Moreover, if the constraint ((**2**), $x \; mod \; 2 = 1$) is added, meaning the restriction to executions where x is odd at **(2)**, then the predicate for **(4)** comes to be *false*, since the *else* branch is never taken. On the other hand, our approach is interested in relaxing the property: slicing on the *parity* of x results in deleting the whole program, since the final x has always the same parity as the input value.

A logical formulation of dependencies for information flow and program slicing can be found in the work by Amtoft and Banerjee [1], where a *logic* proves *independencies* between variables before and after a *prelude* (i.e., a memory transformer).

2 Preliminaries

This section describes the theoretical foundations of this work. The programming framework is outlined, and basic notions of slicing are given. Last section gives an example.

The reader is supposed to be familiar with the basic notions of the *abstract interpretation* [7, 6] theory of *semantic approximation*. Here, it is only pointed out that abstract domains are supposed to be *partitioning*, i.e., closed under set complement. This means that an *upper closure operator* ρ maps minimal sets (i.e., singletons) of concrete values to *atoms* of the domain. This does not imply any loss of generality, since any ρ can be made partitioning by closing it under complement, and the resulting ρ' behaves the same on singletons (i.e., $\rho(\{v_1\}) = \rho(\{v_2\}) \Leftrightarrow \rho'(\{v_1\}) = \rho'(\{v_2\})$). In the following, domains will basically work on singletons.

The framework The language is basically imperative: it includes assignment, sequential composition, conditional and loops, with standard semantics. In addition, following Nielson, Nielson and Hankin [21] (Ch. 2.6), *pointer* expressions are included, which take the form $l ::= x \mid x.sel$, where x is a variable and *sel* is a *selector name*. Memory locations can be addressed, for example, by expressions like *x.cdr*, which remind of Lisp pairs. The syntax comes to be

$$
\begin{array}{rcl}
a & ::= & e \mid b \\
e & ::= & n \mid l \mid e_1\ op_e\ e_2 \mid f_e(\overline{a}) \mid nil \mid new \\
b & ::= & true \mid false \mid \neg b_1 \mid a_1\ op_b\ a_2 \mid f_b(\overline{a}) \\
C & ::= & skip \mid l := a \mid C' \,;\, C'' \mid \\
 & & if\ (b)\ C_t\ else\ C_f \mid while\ (b)\ do\ C_w
\end{array}
$$

Integers and *booleans* are the only primitive types; a pointer l can point to a primitive value or to structured data which can be accessed by selectors. Programs are supposed to be always *well-typed*. *Functions* f_e and f_b take sequences $\overline{a}$ of parameters and return, resp., an integer or a boolean; they do not have *side effects*. Any property will be expressed in terms of pointers (note that a variable is a pointer). Assignment takes the general form $l := a$, where l is a pointer. If l is just a variable, then $l := a$ is an extension of the ordinary assignment of the well-known *while* language. If l contains a selector, then a destructive update of the memory occurs [21]. *new* gives a fresh memory location of suitable type, i.e., $l :=$ *new* allocates memory for all the selectors of l.

A *program state* σ maps pointers to values. The value of l in σ is written $\sigma(l)$, while $[\![C]\!](\sigma)$ is the state obtained by running the command[1] C in σ, and $[\![e]\!](\sigma)$ is the value of an expression e in σ. It is possible to specify *predicates* (logical formulæ) ϕ on states, and $\sigma \models \phi$ means that ϕ holds in σ. Sometimes, *true* is omitted in assertions.

Sharing [25] analysis is needed to keep information about pointers possibly sharing the same location. The result of sharing is available at each program point: (1) $\textsc{sh}(l)$ is the set of all pointers which *may* share with l; and (2) $\textsc{dsh}(l)$ is the set of pointers which *definitely* share with l[2], i.e., are guaranteed to correspond to the same location.

A *program trace* τ is a sequence of program states. τ is the trace of a program C in the state σ if it is the sequence of states obtained by executing C in σ. The set $\tau[p]$, where p is a *program point* in C, contains all the states in τ where the program counter is p (there can be more than one such state if p is inside a loop: one state for every time p is reached).

Program slicing Program slicing [27, 26, 2] was first introduced as *a method used by experienced computer programmers for abstracting from programs. Starting from a subset of the program's behavior, slicing reduces the program to a minimal form which still produces that behavior* [27]. An automatic approach studies how data flow through the program, and computes a minimal[3] subset (the *slice*) of the program which is needed to obtain the desired behavior.

Such behavior is called the *slicing criterion*, and is usually represented, in imperative languages, as a program point and a set of variables, meaning that the slice $\mathcal{P}^s$ of $\mathcal{P}$ w.r.t. the criterion S should be the minimal subprogram of $\mathcal{P}$ with the same behavior on S. More formally, given $S = (p, X)$, for a program point p and a set of variables X, and any input state σ, the *slicing condition* comes to be

$$\forall x \in X.\ [[\![\mathcal{P}^s]\!](\sigma)]_p(x) = [[\![\mathcal{P}]\!](\sigma)]_p(x)$$

where $[[\![\mathcal{P}]\!](\sigma)]_p(x)$ is the value of x in the state which is found at p when executing $\mathcal{P}$ in σ. This is the *static* version of slicing [26], which does not make any assumptions on σ. Anyway, *conditions* can be provided in form of logical formulas, which restrict the set of input states, in the style of conditioned slicing [4] (Section 1).

If the command at p is executed several times, as in a loop, then a sequence of values is obtained. Actually, p is often taken to be the end of the program, without loss of generality, since

- if p is not the end, then assignments *copying* variables in X into fresh variables Y (not modified at any other program points) can be added after p, so that the criterion (p, X) is equivalent to observing Y at the end;
- if p is in a loop, then Y must keep the *sequence* of the values of X. This can be done (1) by using lists and appending the current values of X at every iteration; or (2) without lists, by *encoding* the sequence of values into a natural number and updating it at any iteration.

With these transformations, an equivalent criterion referring to the end of the program can be found for any (p, X), without affecting slicing from the semantic point of view (clearly, issues may arise about practicality and efficiency).

A sound slicing algorithm [27] must only remove commands which are guaranteed not to *interfere* with S in any σ. A typical approach to this problem makes use of *reaching definitions* analysis [14], i.e., computes which assignments can *reach* (i.e., have an effect on) the criterion. Informally, a definition C' reaches another command C'' if a *chain of dependencies* exists between them. Dependencies from C_i to C_{i+1} can be (i) *explicit*, if C_{i+1} uses a variable which is defined by C_i and not redefined on at least one path from C_i to C_{i+1}; or (ii) *implicit*, if C_{i+1} is executed conditionally on the outcome of C_i (e.g., C_i can be a boolean guard). Implicit and explicit dependencies are combined by means of a global computation. In the end, commands are not removed, if they may *reach* the criterion via a chain of dependencies.

A motivating example: append-reverse This example gives an account of abstract program slicing. Lists are defined recursively with selectors *data*, storing the information, and *next*, pointing to the following element. Suppose a property of *well-formedness* be defined on a list, which amounts to having $data = 0$ in the last element. A well-formed empty list is represented as $\langle[0]\rangle$, where

[1] The terms *program* and *command* will be used interchangeably.

[2] If no definite sharing is performed, then $\textsc{dsh}(l) = \{l\}$ can be taken.

[3] A slice does not need to be minimal (actually, the entire program is a slice); anyway, reasonable slicing algorithms are supposed to search for as small a slice as possible.

square brackets indicate that 0 is not a proper element. A correct implementation of *append* has to satisfy, e.g., $append(\langle 1,2,3,4,[0]\rangle, \langle 5,6,[0]\rangle) = \langle 1,2,3,4,5,6,[0]\rangle$, i.e., [0] appears only once at the end of the result.

Consider the following program $\mathcal{P}_{ar}$, which works on two lists *list1* and *list2*, reversing the first one and concatenating it with the second. If $a_1 = \langle 1,2,3,4\rangle$ and $a_2 = \langle 5,6\rangle$, where [0] is left implicit, then $\mathcal{P}$ gives *list2* = $\langle 4,3,2,1,5,6\rangle$.

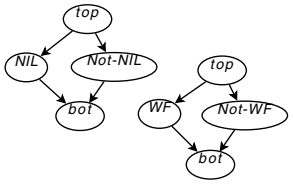
```
list1 := a1 ;  list2 := a2 ;
while (notLast(list1)) {
   tmp := list1.next ;
   list1.next := list2 ;
   list2 := list1 ;
   list1 := tmp ; }
if (nil(list2) ∨ illFormed(list2))
   { res := nil ; } else { res := list2 ; }
```

$$
\begin{array}{lcl}
last(x) & \equiv & notNil(x) \wedge nil(x.next) \\
notLast(x) & \equiv & notNil(x) \wedge notNil(x.next) \\
wellFormed(x) & \equiv & notNil(x) \wedge lastEl(x).data = 0 \\
illFormed(x) & \equiv & notNil(x) \wedge lastEl(x).data \neq 0
\end{array}
$$

In the end, *res* is the concatenation if *list2* is not null nor ill-formed, or *nil* otherwise (since null or ill-formed lists are supposed to be useless for the purpose of this function).

Let the slicing criterion be the final nullity of *res*, i.e., the question q corresponding to the criterion is *is res equal to nil?* and refers to the end of the program. It must be noted that this question is a weaker one w.r.t. the typical slicing question *what is the value of x?*, i.e., the requirement for slicing has been *relaxed* to some property (the nullity) of variables, rather than their exact value.

Abstract domains have to be defined to describe the properties of interest: the picture below shows ρ_{nil} for nullity, and ρ^2_{WF} for well-formedness, which only distinguishes between well-formed lists and all the rest (every abstract value is associated by the $\rightsquigarrow$ notation to a predicate, see above).



$NIL \rightsquigarrow nil$
$Not-NIL \rightsquigarrow notNil$
$WF \rightsquigarrow wellFormed$
$Not-WF \rightsquigarrow \neg\, wellFormed$

It is easy to see that standard slicing on *res* cannot remove any part of the program, since all the code affects *res*. Yet, the outcome of abstract slicing is different.

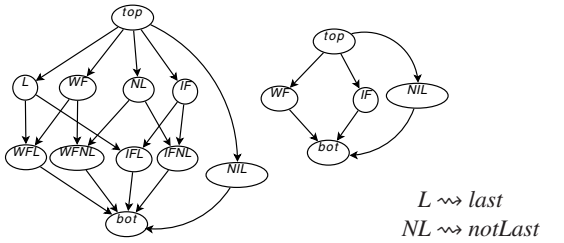
The analysis goes backwards. Let $\mathcal{P}_{ar}$ be written as C_{init} ; C_{loop} ; C_{if}, where the three subprograms are, resp., the initial assignments, the loop and the conditional. The first step is to see that the final conditional is relevant to the criterion, since the nullity of *res* clearly depends on what happens in the branches. An important observation is that, when reaching the loop from below, the initial question q: *is res equal to nil?* is no longer interesting. Instead, the question q' at the program point after the loop should be *is list2 equal to nil or ill-formed?*, which is equivalent to q after the conditional, as a close glance to the code reveals. Generating this q' is crucial in defining abstract slicing.

If ρ^2_{WF} is used for well-formedness, then q' is formulated as *whether list2 has to be abstracted to* WF *or to* NOT-WF. This is the question which has to be considered when analyzing C_{loop}. By looking at the loop semantics, it is possible to see that such property of *list2* does not change, i.e., for every σ before the loop:

$$\rho^2_{\text{WF}}(\sigma(list2)) = \rho^2_{\text{WF}}\left(\left[\!\left[C_{loop}\right]\!\right](\sigma)\,(list2)\right)$$

Since the entire loop is irrelevant to the desired property (i.e., the well-formedness of *list2* after the loop, which amounts to the final nullity of *res*), it can be completely sliced out, thus resulting in a quite different outcome w.r.t. concrete slicing. In fact, in a typical debugging task, there is no need to search in the loop for the *reason* of the ill-formedness of *rest*, since nothing relevant happens there.

Note that the question about the well-formedness of *list2* has been formulated in terms of ρ^2_{WF}. If more precise domains ρ^0_{WF} or ρ^1_{WF} (shown below) are used instead, then the loop cannot be sliced out, as shown in Ex. 6.1.



$L \rightsquigarrow last$
$NL \rightsquigarrow notLast$
$IF \rightsquigarrow illFormed$

In fact, these domains also contain the *nil* abstract value, which is not preserved for *list2* in C_{loop}. I.e., a null value for *list2* may become, after C_{loop}, IF in ρ^1_{WF}, or one between IFL and IFNL in ρ^0_{WF}, thus making the loop *relevant* to the property. In other words, the following holds:

$$\exists\sigma.\ \rho^0_{\text{WF}}(\sigma(list2)) \neq \rho^0_{\text{WF}}\left(\left[\!\left[C_{loop}\right]\!\right](\sigma)\,(list2)\right)$$
$$\exists\sigma.\ \rho^1_{\text{WF}}(\sigma(list2)) \neq \rho^1_{\text{WF}}\left(\left[\!\left[C_{loop}\right]\!\right](\sigma)\,(list2)\right)$$

and forbids to slice out the loop with ρ^0_{WF} or ρ^1_{WF}. This corresponds to the intuition *the weaker the property, the smaller the slice* (Theorem 5.2). The following will show that obtaining *more precise* domains (as, in this case, ρ^0_{WF} or ρ^1_{WF} instead of ρ^2_{WF}) actually comes from a *less precise* analysis, which infers *too strong* preconditions (Ex. 6.1).

3 The underlying theory

This section will formally define the basic semantic notions of abstract slicing: how slicing criteria are defined,

and how (in)dependence is tailored to deal with properties instead of concrete values.

Agreements and slicing criteria An *agreement* is a set of conditions $A_\rho(l)$ for a pointer l and an abstract domain ρ. The definition on traces means that, given a program point p and *two* traces τ_1 and τ_2, an agreement $\mathcal{A}$ requires $\rho(\sigma_1(l)) = \rho(\sigma_2(l))$ for every $A_\rho(l) \in \mathcal{A}$, and every $\sigma_i \in \tau_i[p]$. Note that, for a trace, there can be more than one such state, one for each time execution reaches p.

$$\begin{array}{rcl} \mathcal{A}(\sigma_1, \sigma_2) & \equiv & \forall A_\rho(l) \in \mathcal{A}.\ \rho(\sigma_1(l)) = \rho(\sigma_2(l)) \\ \mathcal{A}(\tau_1, \tau_2, p) & \equiv & \forall \sigma_1 \in \tau_1[p], \sigma_2 \in \tau_2[p].\ \mathcal{A}(\sigma_1, \sigma_2) \end{array}$$

$A_\rho(a)$ refers to the computed value of a: $A_\rho(a)(\sigma_1, \sigma_2) \equiv \rho([\![a]\!](\sigma_1)) = \rho([\![a]\!](\sigma_2))$. Set brackets are often omitted when $\mathcal{A}$ is a singleton. $\mathcal{A}(l)$ is ρ if $A_\rho(l) \in \mathcal{A}$, $\top$ otherwise.

A *slicing criterion* $S \in \mathcal{S}$ is the property of the program which must be preserved when slicing the original $\mathcal{P}$ to obtain a smaller one $\mathcal{P}^s$. In other words, $\mathcal{P}^s$ is the part of $\mathcal{P}$ which is needed to keep S unchanged.

An algorithm for slicing tries to find the smallest subprogram of $\mathcal{P}$ which preserves S. In most frameworks, a criterion is a pair (p, X) where p is a program point and $X \subseteq \mathbb{X}$ is a set of program variables: this means that the property to be preserved is the value of variables in X at p. In abstract slicing, criteria can be specified w.r.t. *abstract properties* (on pointers): the pair $(p, \{A_{\rho_1}(l_1) \ldots A_{\rho_k}(l_k)\})$ means that, for every l_i, the property ρ_i must be preserved in $\mathcal{P}^s$, i.e., that $\rho_i(\sigma(l_i))$ must be the same as $\rho_i(\sigma^s(l_i))$, where σ and σ^s are states of traces of, resp., $\mathcal{P}$ and $\mathcal{P}^s$ at p. Without loss of generality, p is assumed to be the end of the program (Sec. 2), so that it will be left implicit in the following.

Due to this, a slicing criterion takes the same form as an agreement, and, in the following, these concepts will be used somehow interchangeably. 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, where the slicing criterion is an agreement.

Abstract slicing condition Let $\mathcal{P}^s$ be the slice of $\mathcal{P}$ w.r.t. an agreement (criterion) $\mathcal{A}$. In order for $\mathcal{P}^s$ to be correct, $[\![\mathcal{P}]\!](\sigma)$ and $[\![\mathcal{P}^s]\!](\sigma)$ must agree on $\mathcal{A}$ for every initial σ: $\mathcal{A}([\![\mathcal{P}]\!](\sigma), [\![\mathcal{P}^s]\!](\sigma))$.

Note that a *concrete* criterion L (a set of pointers) can be expressed as $\{A_\perp(l)\}_{l \in L}$, meaning that two traces agree on the *exact* value of L, as required by the *identity* domain $\perp$.

Example 3.1 *Consider the code $\mathcal{P}_{ar}$ in Sec. 2: as pointed out before, the slice $\mathcal{P}^s_{ar}$ on the final nullity of res does not include the loop: it takes the form*

```
list2 := a2 ;
if ( nil( list2 ) ∨ illFormed( list2 ) )
{ res := nil; } else { res := list2; }
```

where the first command has been sliced out as well, since it only affects list1 (provided a_2 does not depend on list1). In fact, running both $\mathcal{P}_{ar}$ and $\mathcal{P}^s_{ar}$ on some σ leads to the same value for a_2, so that list2 reaches the conditional with the same value w.r.t. ρ^2_{WF} since the loop is irrelevant to it. Finally, since the guard has the same value, the final nullity of res is the same (since res := list2 is only executed with list2 ≠ nil). Therefore, the slice is correct.

Independence This section defines the independence of an expression w.r.t. an agreement on states, a predicate on states and an output domain on values [20]: $(\mathcal{A})^\phi \twoheadrightarrow (a, \rho)$ means that, for every σ_1 and σ_2 where ϕ holds, $\mathcal{A}(\sigma_1, \sigma_2)$ implies $A_\rho(a)(\sigma_1, \sigma_2)$, i.e., that the results of evaluating a agree on ρ. This definition is similar to narrow non-interference [12] (NANI) on expressions, where domains are actually tuples of domains. The present definition is specialized to the case where there is no public/private distinction, and enriched with restrictions ϕ on states.

Independence states that *input variables do not affect the value of a w.r.t. ρ if their variability obeys ϕ and $\mathcal{A}$.* In other words, it is not required that *all* (concrete) variations of states be irrelevant, but only those which satisfy ϕ and do not make a difference w.r.t. $\mathcal{A}$, i.e., such that the variation *agrees* with the original. Finding $\mathcal{A}$ s.t. $(\mathcal{A})^\phi \twoheadrightarrow (a, \rho)$ amounts to compute the *maximal variability* on the input which does not affect the abstract property of a.

Example 3.2 *Consider the integer expression xyz^2. Let* SIGN *be the abstract domain of sign:* $\text{SIGN}(v_1) = \text{SIGN}(v_2)$ *iff v_1 and v_2 are both negative or both non-negative. In this case, the assertion $(A_{\text{SIGN}}(y))^{x>0} \twoheadrightarrow \left(xyz^2, \text{SIGN}\right)$ holds. In fact, x and z can take any value (with $x > 0$) without affecting the sign of xyz^2 (although x and z can affect its concrete value). On the other hand, the sign of xyz^2 does not change as long as the sign of y does not, but a change in the sign of y* propagates *to the sign of xyz^2.*

4 Inferring information for slicing

This section describes the program analysis steps which are needed in order to compute abstract slices: (1) proving *invariance*, i.e., that executing a command is irrelevant to a given property on data; (2) studying how *agreements* propagate through the program code, in order to find the conditions for states to vary without changes in the criterion. The latter is obtained with a set of rules, the A-system, basically inspired by previous work on abstract non-interference [11].

From the semantic point of view, a command C can be *sliced out* if a property which is strong enough to ensure agreement on the criterion is invariant through the execution of C. In other words, what the command does can only

modify properties which do not make a difference in the desired observation.

Example 4.1 (continued from Ex. 3.1) *The question about the final nullity of res is found, by propagating agreements, to be equivalent to the well-formedness (on ρ^2_{WF}) of list2 after the loop. Therefore, the loop can be sliced out since it does not modify such property.*

On the practical side, deleting C relies on (1) systematically propagating a property which is *weak* (used as the opposite of *strong*, restrictive) enough to be semantically invariant on C; and (2) being able to prove such invariance.

4.1 Inferring invariance

Slicing needs information about properties of data which are preserved through the execution of a command. Such information takes the form of assertions $\text{INV}^{\phi}(\mathcal{A}, C)$ meaning that, for every state σ such that $\sigma \models \phi$, the condition $\mathcal{A}(\sigma, [\![C]\!](\sigma))$ holds. In other words, the effect of C on the program state is irrelevant to $\mathcal{A}$ (i.e., $\mathcal{A}$ is *invariant* on C), so that, to this purpose, C cannot be distinguished by *skip*.

In the following, we assume to have a (sound) static analyzer which answers *yes* to the question *is $\mathcal{A}$ invariant on C under the condition ϕ?* if it is able to guarantee that the assertion $\text{INV}^{\phi}(\mathcal{A}, C)$ holds, and *no* if this guarantee cannot be provided. This is quite a standard abstract interpretation approach to static analysis: the input-output pairs of the *denotational semantics* of a command are abstracted w.r.t. $\mathcal{A}$, and the analyzer tries to detect that any *abstract pair* takes the form (V, V) for some abstract value V, meaning that, at the abstract level, the semantics is the identity function.

The use of ϕ allows to improve the precision by taking *contexts* into account. For example, analyzing a command C inside a loop can get a more precise result if information about the truth value of the loop guard is also considered.

Example 4.2 *Let C be $x := x * y$, and ϕ be $y > 0$. In this case, ϕ is required to successfully answer the question $\text{INV}^{\phi}(\{A_{\text{SIGN}}(x)\}, C)$, since knowing the sign of y guarantees that the sign of x does not change after the assignment.*

4.2 The logic for propagating agreements

This section describes how agreements are propagated via a system of logical rules, the A-system. Hoare-style triples [15] are used for this purpose, in the style of *weakest precondition calculus* [10]. Basically, the precondition is the weakest agreement on two states before a command such that the agreement specified by the post-condition holds after the command. Predicates on program states can be used, so that triples are, actually, 4-tuples which only take into account a subset of the states: $\{\mathcal{A}\}^{\phi}\ C\ \{\mathcal{A}'\}$ (where

$$\frac{\text{INV}^{\phi}(\mathcal{A}, C)}{\{\mathcal{A}\}^{\phi}\ C\ \{\mathcal{A}\}}\ \text{A-INV}$$

$$\frac{}{\{\mathcal{A}\}^{\phi}\ \mathit{skip}\ \{\mathcal{A}\}}\ \text{A-SKIP}$$

$$\frac{\{\mathcal{A}\}^{\phi}\ C\ \{\mathcal{A}'\} \quad \{\mathcal{A}'\}^{C(\phi)}\ C'\ \{\mathcal{A}''\}}{\{\mathcal{A}\}^{\phi}\ C\,;C'\ \{\mathcal{A}''\}}\ \text{A-CONCAT}$$

$$\frac{\{\mathcal{A}_2\}^{\phi_2}\ C\ \left\{\mathcal{A}'_2\right\} \quad \mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \quad \mathcal{A}'_2 \sqsubseteq \mathcal{A}'_1 \quad \phi_1 \Rightarrow \phi_2}{\{\mathcal{A}_1\}^{\phi_1}\ C\ \left\{\mathcal{A}'_1\right\}}\ \text{A-SUB}$$

$$\frac{\left(\begin{array}{c}\forall l_{sh} \in \text{SH}(l)\,.\ (\mathcal{A})^{\phi} \twoheadrightarrow (a, \mathcal{A}'(l_{sh})) \\ \forall l_{sh} \in \text{SH}(l) \setminus \text{DSH}(l)\,.\ \forall \sigma \models \phi. \\ \mathcal{A}'(l_{sh})(\sigma(l_{sh})) = \mathcal{A}'(l_{sh})(([\![a]\!](\sigma))(l_{sh}))\end{array}\right)}{\left\{\mathcal{A}\left[l_{nsh} \leftarrow \mathcal{A}(l_{nsh}) \sqcap \mathcal{A}'(l_{nsh})\right]_{\forall l_{nsh} \notin \text{DSH}(l)}\right\}^{\phi}\ l := a\ \{\mathcal{A}'\}}\ \text{A-ASSIGN}$$

$$\frac{\{\mathcal{A}\}^{\phi}\ C_t \diamond C_f\ \{\mathcal{A}'\}}{\{\mathcal{A}\}^{\phi}\ \mathit{if}\ (b)\ C_t\ \mathit{else}\ C_f\ \{\mathcal{A}'\}}\ \text{A-IF'}$$

$$\frac{\{\mathcal{A}_t\}^{\phi \wedge b}\ C_t\ \{\mathcal{A}\} \qquad \left\{\mathcal{A}_f\right\}^{\phi \wedge \neg b}\ C_f\ \{\mathcal{A}\}}{\left\{\mathcal{A}_b \sqcap \mathcal{A}_t \sqcap \mathcal{A}_f\right\}^{\phi}\ \mathit{if}\ (b)\ C_t\ \mathit{else}\ C_f\ \{\mathcal{A}\}}\ \text{A-IF''}$$

$$\frac{\phi \Rightarrow C_w(\phi) \quad \{\mathcal{A} \sqcap \mathcal{A}_b\}^{\phi \wedge b}\ C_w\ \{\mathcal{A} \sqcap \mathcal{A}_b\}}{\{\mathcal{A} \sqcap \mathcal{A}_b\}^{\phi}\ \mathit{while}\ (b)\ \mathit{do}\ C_w\ \{\mathcal{A} \sqcap \mathcal{A}_b\}}\ \text{A-WHILE}$$

Figure 1. The A-system

the *true* predicate is often omitted) holds if, for every σ_1 and σ_2,

$$\sigma_1 \models \phi \ \wedge\ \sigma_2 \models \phi \ \wedge\ \mathcal{A}(\sigma_1, \sigma_2) \ \Rightarrow\ \mathcal{A}'([\![C]\!](\sigma_1), [\![C]\!](\sigma_2))$$

The *transformed predicate* $C(\phi)$ is one which is guaranteed to hold after a command C, given that ϕ holds before, in the style of *strongest post-condition calculus* [3].

The A-system (Fig. 1) is related to recent work on narrow non-interference [11]. Such work defines a similar system of rules, the N-rules, for assertions $[\eta]C(\eta')$, where η and η' are basically the (tuples of) abstract domains corresponding to, resp., $\mathcal{A}$ and $\mathcal{A}'$. The systems differ in that:

- pointers require the A-rule for assignment to account for sharing, while N-rules only work on integers;
- in the present approach, partitions are implicit since domains are supposed to be partitioning;
- the A-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 ϕ on program states, which can improve the precision, are not supported;

A-INV This rule makes the relation between invariance and the A-system clear. The triple $\{\mathcal{A}\}\ C\ \{\mathcal{A}\}$ amounts to say that two traces agree *after* C, provided they agree *before* on the same $\mathcal{A}$. On the other hand, the invariance of $\mathcal{A}$ on C means that any state *before* C agrees on $\mathcal{A}$ with the state *after* C. Invariance is a stronger requirement than the mere preservation $\{\mathcal{A}\}\ C\ \{\mathcal{A}\}$ of agreements.

Example 4.3 *Let parity be the property of interest:* $\text{PAR}(v_1) = \text{PAR}(v_2)$ *iff* v_1 *and* v_2 *have the same parity. In this case,* $x := x + 1$ *does not preserve* $\text{PAR}(x)$*, but two initial states agreeing on* $\text{PAR}(x)$ *lead to final states which still agree on it. Therefore,* $\{A_{\text{PAR}}(x)\}\ x := x + 1\ \{A_{\text{PAR}}(x)\}$ *holds. On the other hand,* $x := x + 2$ *also satisfies a stronger requirement: that* $\text{PAR}(x)$ *does not change. Therefore, besides having* $\{A_{\text{PAR}}(x)\}\ x := x + 2\ \{A_{\text{PAR}}(x)\}$*, the equality* $A_{\text{PAR}}(x)(\sigma, [\![x := x + 2]\!](\sigma))$*, equivalent to* $\text{INV}(A_{\text{PAR}}(x), x := x + 2)$*, is also true.*

A-SKIP, A-CONCAT, A-SUB The A-SKIP rule describes *no-op*. The assertion holds for every $\mathcal{A}$ and ϕ since $[\![skip]\!](\sigma) = \sigma$.

A-CONCAT is also easy: soundness holds by transitivity.

In A-SUB, $\sqsubseteq$ stands for the (pointwise) comparison on agreements, i.e., $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if $\forall l. \mathcal{A}_1(l) \sqsubseteq \mathcal{A}_2(l)$, where $\mathcal{A}_1(l) \sqsubseteq \mathcal{A}_2(l)$ is the comparison on the *precision* of abstract domains, meaning that $\mathcal{A}_1(l)$ is *more precise* than $\mathcal{A}_2(l)$.

A-ASSIGN This rule means: if some $\mathcal{A}$ excludes (when ϕ holds) the dependence w.r.t. $\mathcal{A}'$ of a on l and all pointers possibly sharing with it, then such $\mathcal{A} \sqcap \mathcal{A}'$ is strong enough as a precondition. However the use of $\text{DSH}(l)$ is meant to increase the precision (i.e., to weaken the precondition), since the value of any $l_{dsh} \in \text{DSH}(l)$ is lost after the assignment, unless a depends on it. Consequently, the precondition $\mathcal{A}_{pre}$ can have, as $\mathcal{A}_{pre}(l_{nsh})$, the domain $\mathcal{A}(l_{nsh})$ instead of $\mathcal{A}'(l_{nsh}) \sqcap \mathcal{A}(l_{nsh})$, still preserving correctness. Taking $\mathcal{A}_{pre} = \mathcal{A} \sqcap \mathcal{A}'$ is correct, but less precise.

The second condition of the rule requires the assignment be irrelevant w.r.t. $\mathcal{A}'$ for pointers which *may* be updated or left unchanged. This could look as quite conservative, but is really needed for correctness.

The A-ASSIGN rule can be compared with the rule N3 for assignment in the N-system [12]:

$$\frac{[\eta]a(\rho),\ [\Pi(\eta_y) \sqsubseteq \Pi(\rho_y)]_{y \in \mathtt{L} \setminus \{x\}},\ x \in \mathtt{L}}{[\eta]x := a(\rho)}\ \text{N3}$$

where $[\eta]e(\rho)$ means

$$\forall \sigma_1, \sigma_2.\ \eta(\sigma_1^{\mathtt{L}}) = \eta(\sigma_2^{\mathtt{L}}) \ \Rightarrow\ \rho([\![a]\!](\sigma_1)^{\mathtt{L}}) = \rho([\![a]\!](\sigma_2)^{\mathtt{L}})$$

and (1) ρ is, actually, $\mathcal{A}'(l_{sh})$ in the condition of A-ASSIGN; (2) $\sigma^{\mathtt{L}}$ restricts σ to public variables; and (3) $\Pi(\eta_y) \sqsubseteq \Pi(\rho_y)$ means that the *partition* on singletons induced by η_y (the component of η corresponding to y) must be more concrete than the one of ρ_y. Rules N3 and A-ASSIGN differ in that:

- pointers require A-ASSIGN to account for sharing;
- in A-ASSIGN, partitions are kept implicit since domains are partitioning, and the $\sqsubseteq$ condition on η_y and ρ_y is guaranteed by the fact that $\mathcal{A}_{pre}$ is stronger than $\mathcal{A}$;
- $x \in \mathtt{L}$ does not appear in A-ASSIGN since there are no private variables; also, states are not restricted to L.

Example 4.4 (continued from Sec. 2) *Consider the assignments in the branches of* C_{if}*, with* $\{A_{\rho_{nil}}(res)\}$ *as post-condition.* $res := nil$ *satisfies* $(\emptyset) \twoheadrightarrow \left(nil, \{A_{\rho_{nil}}(res)\}(res)\right)$*, i.e.,* $(\emptyset) \twoheadrightarrow (nil, \rho_{nil})$*, and res does not share (and* $\text{SH}(res) \setminus \text{DSH}(res) = \emptyset$*). In this case,* $\emptyset$ *(i.e., no restrictions) can be taken as precondition, since* $res \in \text{DSH}(res)$*, so that its value does not go through the assignment. The resulting triple comes to be*

$$\{\emptyset\}\ res := nil\ \left\{A_{\rho_{nil}}(res)\right\}$$

which makes sense since any final state agrees on $\rho_{nil}(res)$*. On the other hand,* $res := list2$ *yields*

$$\left\{A_{\rho_{nil}}(list2)\right\}\ res := list2\ \left\{A_{\rho_{nil}}(res)\right\}$$

since the nullity of res after the command is equivalent to the nullity of list2 before.

Lemma 4.1 (soundness of A-ASSIGN) *Let* $\mathcal{A}_{pre}(\sigma_1, \sigma_2)$*, and* $\sigma_i' = [\![l := a]\!](\sigma_i)$*. Then, provided* $\sigma_1 \models \phi$ *and* $\sigma_2 \models \phi$ *and the conditions of the rule hold,* $\mathcal{A}'(\sigma_1', \sigma_2')$ *holds, i.e.,*

$$\forall l_0.\ \mathcal{A}'(l_0)(\sigma_1'(l_0)) = \mathcal{A}'(l_0)(\sigma_2'(l_0))$$

Proof All the proofs can be found in an extended (same text plus proofs) technical report version [29].

A-IF In a conditional *if* (b) C_t *else* C_f, there are two possibilities. Rule A-IF' states that an input agreement which implies the output one, regardless of the path taken, is a good candidate as a precondition. Here, the assertion $\{\mathcal{A}\}^{\phi}\ C' \diamond C''\ \{\mathcal{A}'\}$ means that

$$\forall \sigma_1, \sigma_2.\quad \mathcal{A}(\sigma_1, \sigma_2) \wedge \sigma_1 \models \phi \wedge \sigma_2 \models \phi \quad \Rightarrow$$
$$\mathcal{A}'([\![C']\!](\sigma_1), [\![C']\!](\sigma_2), [\![C'']\!](\sigma_1), [\![C'']\!](\sigma_2))$$

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

Note that such $\mathcal{A}$ can always be found (in the worst case, it is the identity $\{A_{\perp}(l)\}_l$). However, sometimes it can be more convenient to exploit information about b. In such cases, A-IF" can be applied, which means that the initial agreement $\mathcal{A}_t \sqcap \mathcal{A}_f$ is strong enough to verify the final one, provided the same branch is taken in both traces, as $\mathcal{A}_b$ requires. In fact, $\mathcal{A}_b$ is built from b, and distinguishes states w.r.t. its value:

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

The rule means that, whenever two states agree on the branch to be executed, and the triples on the branches hold, the whole triple holds as well. Knowing the value of b when analyzing the branches may allow to obtain a better result.

Example 4.5 (continued from Sec. 2 and Ex. 4.4) *In C_{if}, $\mathcal{A}_b$ comes to be $\{A_{\rho^2_{\text{WF}}}(list2)\}$, since the abstract domain formalizes exactly the condition in the guard:*

$$\rho^2_{\text{WF}}(\sigma(list2)) = \text{WF} \quad \Leftrightarrow \quad \neg\,([\![b]\!]\,(\sigma))$$

The precondition obtained by A-IF" *from results in Ex. 4.4 is*

$$\left\{A_{\rho^2_{\text{WF}}}(list2)\right\} \sqcap \emptyset \sqcap \left\{A_{\rho_{nil}}(list2)\right\} = \left\{A_{\rho^2_{\text{WF}}}(list2), A_{\rho_{nil}}(list2)\right\}$$

However, information about the context can be used in the else branch (namely, that b is false, which implies list2 ≠ nil), so that the triple for res := list2 becomes

$$\{\emptyset\}^{\neg b}\ \ res := list2\ \ \left\{A_{\rho_{nil}}(res)\right\}$$

and leads to a more precise assertion for C_{if}:

$$\left\{A_{\rho^2_{\text{WF}}}(list2)\right\}\ C_{if}\ \left\{A_{\rho_{nil}}(res)\right\}$$

Lemma 4.2 (soundness of A-IF") *If σ_1 and σ_2 satisfy ϕ and agree on $\mathcal{A}_b \sqcap \mathcal{A}_t \sqcap \mathcal{A}_f$, then the corresponding outputs σ'_1 and σ'_2 agree on $\mathcal{A}$ under the hypotheses of the rule.*

Note that the rule to be chosen for the conditional depends on the precision of the outcome: A-IF" can be a good choice if (1) it can be applied; and (2) its result is better (weaker) than the one obtained by A-IF'.

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

Lemma 4.3 (soundness of A-WHILE) *Let σ^0_1 and σ^0_2 satisfy ϕ, and agree on $\mathcal{A} \sqcap \mathcal{A}_b$. Then, given $\sigma'_i = [\![while\ (b)\ do\ C_w]\!]\left(\sigma^0_i\right)$, the result $(\mathcal{A} \sqcap \mathcal{A}_b)(\sigma'_1, \sigma'_2)$ holds.*

Theorem 4.4 (A-soundness) *Let C be a command, $\mathcal{A}'$ be required after C, ϕ be a predicate and p be the program point before C. Let also $\mathcal{A}$ be an agreement computed before C by means of the A-system. Let τ_1 and τ_2 be two traces, and the states $\sigma_1 \in \tau_1[p]$ and $\sigma_2 \in \tau_2[p]$ satisfy $\mathcal{A}(\sigma_1, \sigma_2)$ and ϕ. Then, the condition $\mathcal{A}'(\sigma'_1, \sigma'_2)$ holds, where $\sigma'_i = [\![C]\!]\,(\sigma_i)$.*

5 Slicing a program

This section defines a compositional function SLICE, with an auxiliary SLICE′, for slicing a program w.r.t. an abstract slicing criterion and a predicate on states, using the results obtained by invariance analysis and the A-system. The function SLICE takes a command C, a criterion $\mathcal{A}$ and a predicate ϕ on states, and returns the slice[4] of C w.r.t. $\mathcal{A}$ and ϕ. If, in the initial call, ϕ is not the true predicate, then the algorithm implements a conditioned [4] form of abstract slicing (Section 1).

$$\begin{array}{rcll}
\text{SLICE}\,(C, \mathcal{A})^{\phi} & = & \textit{skip} & \textit{if } \text{INV}^{\phi}\,(\mathcal{A}, C) \\
& = & \text{SLICE}'\,(C, \mathcal{A})^{\phi} & \textit{otherwise}
\end{array}$$

$$\begin{array}{rcl}
\text{SLICE}'\,(l := a, \mathcal{A}')^{\phi} & = & l := a \\
\text{SLICE}'\,(C'\,;\,C'', \mathcal{A}'')^{\phi} & = & C'^{(s)}\,;\,C''^{(s)} \\
\textit{where} & & \phi' = C'(\phi) \vee C'^{(s)}(\phi) \\
& & \{\mathcal{A}'\}^{\phi'}\ \ C''\ \ \{\mathcal{A}''\} \\
& & C'^{(s)} = \text{SLICE}\,(C', \mathcal{A}')^{\phi} \\
& & C''^{(s)} = \text{SLICE}\,(C'', \mathcal{A}'')^{\phi'} \\
\text{SLICE}'\left(\textit{if } (b)\ C_t \textit{ else } C_f, \mathcal{A}'\right)^{\phi} & = & \textit{if } (b)\ C^s_t \textit{ else } C^s_f \\
\textit{where} & & C^s_t = \text{SLICE}\,(C_t, \mathcal{A}')^{\phi \wedge b} \\
& & C^s_f = \text{SLICE}\left(C_f, \mathcal{A}'\right)^{\phi \wedge \neg b} \\
\text{SLICE}'\,(\textit{while } (b) \textit{ do } C_w, \mathcal{A}')^{\phi} & = & \textit{while } (b) \textit{ do } C^s_w \\
\textit{if} & & \{\mathcal{A}\}^{\phi \wedge b}\ \ C_w\ \ \{\mathcal{A}'\} \\
& & \mathcal{A} \sqsubseteq \mathcal{A}' \sqsubseteq \mathcal{A}_b \\
\textit{where} & & C^s_w = \text{SLICE}\,(C_w, \mathcal{A})^{\phi \wedge b}
\end{array}$$

The basic meaning is: when SLICE is given a program C, an agreement $\mathcal{A}$ and a predicate ϕ, it tries first to slice C out completely by proving that C preserves $\mathcal{A}$ given ϕ. Otherwise, it goes recursively into the program structure, trying to slice out some sub-parts. For example, $\text{SLICE}\,(l := a, \mathcal{A})^{\phi}$ has two possible outcomes: (i) *skip*, if $\mathcal{A}$ is invariant on the assignment; or (ii) $l := a$, otherwise (because SLICE′ has been called). Most rules are easy to understand.

The rule for concatenation relies on finding two triples for the sub-commands, and slice them according to the agreement which is found to hold between C' and C'. Note that $C'^{(s)}(\phi)$ is not available yet when computing the triple on C''. This problem must be carefully addressed in an algorithmic approach, for example by computing a *fixpoint* which progressively refines the predicate and the slice.

The general meaning of the rules for loops is that a loop can be (1) completely removed, if $\text{INV}^{\phi}\,(\mathcal{A}, C)$; (2) kept as a loop and had the body sliced, if the slice of the body does not change neither $\mathcal{A}$ w.r.t. the original body, nor the number of iterations. Note that such $\mathcal{A}$ can always be found: in the worst case, it is the identity $\{A_{\perp}(l)\}_l$.

The function SLICE is called several times at different program points. Due to this, every p (apart from those which are internal to removed code) can be seen to be *annotated* with agreements: p is annotated with $\mathcal{A}$ (written $p : \mathcal{A}$) if $\text{SLICE}\,(C, \mathcal{A})^{\phi}$ is called, and C ends at p (it is easy to see that there is only one call for every p). Due to the rule on concatenation, a command C between program points p and p', with $p : \mathcal{A}$ and $p' : \mathcal{A}'$, satisfies $\{\mathcal{A}\}^{\phi}\ \ C\ \ \{\mathcal{A}'\}$, if the call was $\text{SLICE}\,(C, \mathcal{A}')^{\phi}$. The beginning of the program is annotated with $\mathcal{A}_0$ s.t. $\{\mathcal{A}_0\}^{\phi}\ \ C_0\ \ \{\mathcal{A}\}$, where C_0 is the first command, and $\text{SLICE}\,(C_0, \mathcal{A})^{\phi}$ has been called.

[4]Note that, here, statements are replaced by *skip* instead of being removed; however, a final removal of all *skip* is trivial.

Back to append-reverse This section resumes the discussion in Sec. 2 and gives the result of applying our technique to $\mathcal{P}_{ar}$. The initial call is $\textsc{slice}\left(\mathcal{P}_{ar}, A_{\rho_{nil}}(res)\right)^{true}$: since the nullity of *res* is clearly not invariant on $\mathcal{P}_{ar}$, $\textsc{slice}'\left(\mathcal{P}_{ar}, A_{\rho_{nil}}(res)\right)^{true}$ is called.

The rule for concatenation applies to C_0 ; c_{if}, where C_0 is C_{init} ; C_{loop}. It finds (1) $\phi' = C_0(true) \vee C_0^s(true) = true$ (here, C_0^s is not needed, but see Sec. 5); (2) $\left\{A_{\rho^2_{\text{WF}}}(list2)\right\}^{true}$ C_{if} $\left\{A_{\rho_{nil}}(res)\right\}$ (see Ex. 4.5); (3) $C_0^s = \textsc{slice}\left(C_0, A_{\rho^2_{\text{WF}}}(list2)\right)^{true}$ (see below); and (4) $C_{if}^s = \textsc{slice}\left(C_{if}, A_{\rho_{nil}}(res)\right)^{true} = C_{if}$ (i.e., there is no slicing here).

As for $\textsc{slice}\left(C_{init}\,;C_{loop}, A_{\rho^2_{\text{WF}}}(list2)\right)^{true}$, the same rule applies. Again, $\phi' = true$. In this case, the call $\textsc{slice}\left(C_{loop}, A_{\rho^2_{\text{WF}}}(list2)\right)^{true}$ takes the first pattern because of invariance, and *skip* is returned. In the concatenation rule, $A_{\rho^2_{\text{WF}}}(list2)$ is also given as the agreement before C_{loop}, so that the following call will be $\textsc{slice}\left(C_{init}, A_{\rho^2_{\text{WF}}}(list2)\right)^{true}$, which slices out $list1 := a_1$ and keeps $list2 := a_2$.

The main achievement is that the whole loop can be excluded from the slice, since it is not responsible for the well-formedness of *list2*. In general, this can have a big impact on debugging the program, e.g., when the focus is on why *res* = *nil* at the end. Note that standard slicing would consider the whole program as relevant.

Correctness This theorem proves that slices satisfy the abstract slicing condition. Actually, a stronger condition holds, since inputs are only required to agree on $\mathcal{A}_0$, instead of being equal.

Theorem 5.1 *Let $C^s = \textsc{slice}\,(C, \mathcal{A})^{\phi}$, and σ and σ^s be two states satisfying $\mathcal{A}_0(\sigma, \sigma^s)$, $\sigma \models \phi$ and $\sigma^s \models \phi$, where the initial program point is annotated with $\mathcal{A}_0$. Then, $\mathcal{A}([\![C]\!](\sigma), [\![C^s]\!](\sigma^s))$ holds.*

The correctness of SLICE shows that it makes sense to use an agreement as a slicing criterion. In fact, a criterion can be seen as an agreement on traces which correspond to two programs C and C^s which are different but tightly related by a *command erasure* transformation.

Another important property is that slices become smaller if criteria become weaker. Let $C_1 \leq C_2$ hold if C_1 is obtained from C_2 by replacing some commands by *skip* (which boils down to be a slice of C_2).

Theorem 5.2 *Let $\phi_1 \Rightarrow \phi_2$ and $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$. Then, $C^{s1} \leq C^{s2}$, where $C^{s1} = \textsc{slice}\,(C, \mathcal{A}_1)^{\phi_1}$ and $C^{s2} = \textsc{slice}\,(C, \mathcal{A}_2)^{\phi_2}$.*

6 Practical issues

The SLICE algorithm is not meant to be directly executable. Rather, it is more to be seen as a *systematic semantic approach* to abstract slicing. Several components are needed in order to answer crucial questions:

1. an *invariance* analyzer to prove assertions $\textsc{inv}^{\phi}(\mathcal{A}, C)$;
2. *weakest precondition calculus* [10] to find $\mathcal{A}$ s.t., for given C, $\mathcal{A}'$ and ϕ, the assertion $\{\mathcal{A}\}^{\phi}\ C\ \{\mathcal{A}'\}$ holds;
3. *symbolic execution* [18] to deal with predicates on states, and *strongest post-condition calculus* [3] to find ϕ' such that, given C and ϕ, the assertion $\sigma \models \phi$ implies $[\![C]\!](\sigma) \models \phi'$ for every σ.

The A-system is a reasonable proposal to answer question 2 but, of course, it is not complete and can be improved. This task and question 1 rely on *sharing* analysis and on computing *independence* of expressions [20]. Not surprisingly, both tasks need non-trivial machinery to symbolically deal with operations on the *abstract domains*.

Besides (see Sec. 5), the rule for concatenation needs to be dealt with carefully in order to be correctly implemented.

Moreover, question 3 can be approached by means of some strongest post-condition calculus. Symbolic execution has already been practically used in computing predicates on states in program slicing: related work (Section 1 includes a discussion of the *conditioned program slicing* [4] framework, which has already been implemented in the ConSIT tool [9].

The precision of the slicing basically depends on how precisely these tasks are solved. In fact, an imprecise outcome of some component (e.g., too strong preconditions, or failure to prove invariance) may result in the impossibility to slice out some parts of the code which are semantically irrelevant to the abstract criterion.

Example 6.1 *In $\mathcal{P}$, the agreement $\left\{A_{\rho^2_{\text{WF}}}(list2)\right\}$ is produced before C_{if}. This means that the* A*-system was clever enough to detect that the boolean guard splits states exactly as ρ^2_{WF} does, and to exploit the non-nullity of list2 in the* else *branch. On the other hand, simply proving less precise results as $\left\{A_{\rho^1_{\text{WF}}}(list2)\right\}$ does not allow to remove the loop.*

7 Conclusions and future work

The present paper introduces a semantic basis for an abstract program slicing algorithm. The proposed technique allows to slice a program with respect to a given property, represented as an abstraction, instead of concrete values.

This kind of reasoning inherently relies on program semantics. Indeed, considering syntax alone is quite a good approximation in the case of concrete slicing, but becomes too imprecise when abstract properties are considered [20]. The theoretical basis is proven to be sound. Implementing the algorithm depends on having static analysis components which are designed to prove assertions on the program semantics. The more assertions can be guaranteed, the better the result of the slicing (i.e., the smaller the abstract slice).

Future work One direction of future work consists of accounting for more realistic frameworks. An *interprocedural* (where procedures can have side effects) formulation would be a first step in this direction. In the longer run, work will be focused on full *Object-Oriented* languages.

The *logical* and *static analysis* components needed to implement the algorithm deserve further study. The power of such techniques from the semantic point of view has to be investigated. In addition, attention will be paid to existing tools which can be used (as they are, or optimized/specialized or generalized) to this purpose.

Finally, effort will be put on *implementing* the presented framework, based on the issues pointed out in Section 6. This is a more advanced task, and needs good solutions to be found for technical/practical issues. Dealing with the symbolic computations involved in the algorithm (e.g., proving abstract independence on expressions) is needed: this is quite a common issue in practical works on abstract non-interference [12, 28].

Acknowledgments

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

Special thanks go to S. Genaim, R. Giacobazzi and I. Mastroeni for the fruitful discussions which helped in developing these ideas and writing this paper.

References

[1] T. Amtoft and A. Banerjee. A Logic for Information Flow Analysis with an Application to Forward Slicing of Simple Imperative Programs. *Science of Comp. Programming*, 64(1), 2007.

[2] D. B. and. Program slicing. *Advances in Computers*, 43, 1996.

[3] E. D. and. *Predicate calculus and program semantics*. 1990.

[4] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned Program Slicing. *Information and Software Technology*, 40, 1998.

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

[6] P. Cousot. Verification by Abstract Interpretation. In *Proc. Int. Symp. on Verification*, 2003.

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

[8] S. Danicic, D. Binkley, T. Gyimóthy, M. Harman, A. Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Science of Comp. Programming*, 62(3), 2006.

[9] S. Danicic, C. Fox, M. Harman, and R. Hierons. ConSIT: A conditioned program slicer. In *Proc. ICSM*, 2000.

[10] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. of the ACM*, 18(8), 1975.

[11] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In N. Jones and X. Leroy, editors, *Proc. POPL*, 2004.

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

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

[14] M. Hecht. *Flow analysis of computer programs*. 1977.

[15] C. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10), 1969.

[16] H. Hong, I. Lee, and O. Sokolsky. Abstract Slicing: A new approach to program slicing based on abstract interpretation and model checking. In J. Krinke and G. Antoniol, editors, *Proc. SCAM*, 2005.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1), 1990.

[18] J. King. Symbolic execution and program testing. *Comm. of the ACM*, 19(7), 1976.

[19] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3), 1988.

[20] I. Mastroeni and D. Zanardini. Data Dependencies and Program Slicing: from Syntax to Abstract Semantics. In *Proc. PEPM*, 2008.

[21] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. 1999.

[22] X. Rival. Abstract dependences for alarm diagnosis. In K. Yi, editor, *Proc. APLAS*, volume 3780 of *LNCS*, 2005.

[23] X. Rival. *Traces Abstraction in Static Analysis and Program Transformation*. PhD thesis, Computer Science Department, École Normale Supérieure, 2005.

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

[25] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. SAS*, volume 3672 of *LNCS*, 2005.

[26] F. Tip. A survey of program slicing techniques. Technical report, CWI (Centre for Mathematics and Computer Science), 1994.

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

[28] D. Zanardini. Higher-Order Abstract Non-Interference. In P. Urzyczyn, editor, *Proc. TLCA*, volume 3461 of *LNCS*, 2005.

[29] D. Zanardini. The Semantics of Abstract Program Slicing. Technical Report CLIP4/2008.0, Technical University of Madrid (UPM), 2008.