A proof system for Abstract Non-Interference

Roberto Giacobazzi and Isabella Mastroeni
Dipartimento di Informatica - Università di Verona
(roberto.giacobazzi|isabella.mastroeni)@univr.it

August 5, 2009

Abstract

In this paper, we provide an inductive proof system for a notion of abstract non-interference which fits in every field of computer science where we are interested in observing how different programs data interfere with each other. The idea is to abstract from language-based security and consider generically data as distinguished between internal (that has to be protected by the program) and observable. In this more general context we derive a proof system which allows us to characterise abstract non-interference properties inductively on the syntactic structure of programs. We finally show how this framework can be instantiated to language-based security.

Keywords: Abstract interpretation, abstract domains, non-interference, closure operators, semantics, static program analysis, logic of programs, verification.

1 Introduction

Information-flow analysis is a fundamental kind of analysis in several fields of computer science. It is essential in code debugging, program analysis, program transformation, and software verification. To understand how information flows in programs means to model the properties of control and data that are transformed dynamically at run-time. Program slicing needs information-flow analysis for understanding which parts of the code are independent; code debugging and testing need models of information-flow for understanding error propagation, language-based security needs information-flow analysis for analysing how confidential data flows due to erroneous or malicious attacks while data are processed by programs. Information-flow analyses are based on the characterisations of the degree of independence between program objects, such as program variables and statements. This is the basic feature of the notion of *dependency* introduced by Cohen [5] and then called *non-interference* and extended in the context of language-based security by Goguen and Meseguer [17]. Language-based security, characterising security policies and models, provides, in this

perspective, an important application field for transposing notions, developed for security models, to information-flow software analysis and applications. The standard way to protect confidential data in security is *access control*: Higher privileges are required in order to access files containing confidential data. It is well known that access control checks do not put constraints on how the information is propagated. Once information is released from its container, the accessing program may, either by mistake or on purpose, improperly transmits the information in some form. In this case, in order to ensure that information can be used only according to specific security rules, also known as information-flow policies, it is necessary to analyze how information flows in program's semantics. Clearly, when computations on public data are non-interfering with those on private resources, no leakage of confidential information is possible by observing public input/output behavior. This principle is a common pattern for specifying security polices in language-based security [23]. Most methods and techniques for checking *secure information flows* in software, ranging from standard data-flow/control-flow analysis techniques to type inference, are based on non-interference. All of these approaches are devoted to prove that a system as a whole, or parts of it, does not allow confidential data to flow towards public variables. Type-based approaches are designed in such a way that well-typed programs do not leak secrets. In a security-typed language, a type is inductively associated at compile-time with program statements in such a way that any statement showing a potential flow disclosing secrets is rejected [25, 27, 29]. Similarly, data-flow/control-flow analysis techniques are devoted to statically discover flows of secret data into public variables [4, 18, 19, 24]. Non-interference is a standard approach to confidentiality problems, and it is based on a characterization of the attacker that does not impose any observational or complexity restriction on the attackers' power. This means that, in this model, the attackers have *full power*, namely they are modeled without any limitation in their quest to obtain confidential information. For this reason non-interference, as defined in the literature, is an extremely restrictive policy. The problem of refining this kind of security policies has been addressed by many authors as a major challenge in language-based information-flow security [23]. Refining security policies means weakening standard non-interference checks, in such a way that these restrictions can be used in practice or can reveal more information about how information flows in programs.

In the literature, we can find mainly two different approaches for weakening non-interference: By constraining the power of the attacker (from the observational or the computational point of view), or by allowing some confidential information to flow (the so called *declassification*). There are several works dealing with both these approaches, but to the best of our knowledge, only one of these can characterize, at the same time, both the power of the attacker's model and the private information that can flow: *abstract non-interference* [12, 15]. Such a model allows us to understand the intuitive relation existing between the attacker's model and the information released: The more powerful the attacker is, the less information can be kept private [14, 16]. Abstract non-interference captures a weaker form of non-interference, where non-interference is made para-

metric relatively to some abstract property of input/output behaviour. Consider the following program written in a simple imperative language, where the **while**-statement iterates until x_1 is 0. Suppose x_1 is a confidential variable and x_2 is a public variable:

$$\textbf{while } x_1 > 0 \textbf{ do } x_2 := x_2 + 2;\ x_1 := x_1 - 1 \textbf{ endw}$$

In standard non-interference there is an implicit flow from x_1 to x_2, due to the **while**-statement, since x_2 changes depending on the initial value of x_1. This represents the case where no restriction is considered on the power of the attacker. However, suppose that the attacker can observe only the parity of public integer variables (x_2). It is clear that this property cannot be changed by the execution of the program. This means that there's no information-flow from private to public if the attacker can only observe parity. Abstract non-interference generalizes this idea to arbitrary abstractions of programming language semantics and to arbitrary contexts, where the non-interference analysis is fundamental. This provides both a characterization of the degree of dependency between different components of a program, relatively to what an observer can analyze about its input/output information flow, and the possibility to certify code relatively to some weaker form of non-interference.

This problem has been attacked first by Cohen [6]. In his definition of *selective dependency*, he considers more general situations, where only a *portion* of private information affects the observable data; as it happens in the following example:

$$l := |l| * \mathit{Sign}(h)$$

where $|l|$ is the absolute value of the public variable l, while $\mathit{Sign}(h)$ is the sign of the private variable h. In this case only the sign of h has effect on the value of l. Hence, if we do not have any restriction on the observational power of the attacker, then we can conclude that only the sign of the private input can be detected, since it is the only portion of private data that flows in the public output. Moreover, if the attacker can only observe the absolute value of public data, then this assignment is secure. These considerations suggested a notion of non-interference where it is also possible to characterize which portion of confidential data interferes with the observable output. This requires a considerable extension of Cohen's original approach by selective dependencies. For instance the two characterizations above are combined in such a way that the program fragment

$$l := l * h^2$$

can be certified as secure if the attacker can only observe the parity of the public variable l and if we are interested only in keeping private the sign of the private variable h. In this expression, it is the semantics of the program that creates a implicit *semantic firewall* between public and private variables that protects the sign of h. Therefore, in the more general context, given the observer's model, abstract non-interference allows us to characterise, not only *if* there is an information-flow, but also *what* is flowing, when it turns out that the program violates non-interference.

The problem. Abstract non-interference is based on the general idea that data are distinguished into two classes: what is observable (public in the security context) and what has to remain *internal* to the program (private in security). This classification is parametric on the model of an observer, which is an abstract interpretation of the program semantics. A program satisfies the abstract non-interference condition relatively to some given abstraction (observer) if the abstraction obfuscates any possible interference between internal and observable data. In [12, 15] the authors introduce a step-by-step weakening of Goguen and Meseguer's non-interference by specifying abstract non-interference as a property of the program semantics. The idea of modeling observers as abstract domains provides advanced methods for deriving these observers by systematically transforming the corresponding abstract domains. An algebraic characterization of the most precise harmless observer, i.e., the most precise abstraction for which the program satisfies the abstract non-interference property, is given as a fixpoint domain construction. This abstraction, as well as any abstraction for which the program satisfies abstract non-interference, is both a model of an harmless observer and a certificate for the non-interference degree of the program. The problem we want to investigate is how it is possible to *compose* abstract non-interference certificates. We would like to make this composition systematic, and therefore we aim to characterise a proof system, inductive on the syntactic structure of programs. This proof system allows the direct derivation of abstract non-interference certificates only for elementary statements and then to obtain more complex certificates by using the rules of the proof system. In this way, we can think of using abstract non-interference in automatic program certification mechanisms, such as in proof-carrying code architectures [21] and in type-based verification algorithms.
The logical approach to secure information flow is not new. In [10] dynamic logic is used for characterizing secure information flows, deriving a theorem prover for checking programs. In [1] an axiomatic approach for checking secure information flows is provided. In particular the authors syntactically derive the secure information flows that may happen during the execution. Both these works don't characterize the power of the attacker.

Main contribution and structure of the paper. The aim of this paper is to provide a compositional proof system for certifying abstract non-interference in programming languages. In this way we can prove, inductively on the syntactic structure of programs, properties of abstract non-interference relatively to some given abstraction of its input/output. Abstractions are specified in the standard abstract interpretation [8] framework. The proof systems is based on the derivation of abstract non-interference assertions, which specify the non-interference degree of a program relatively to a given model of attacker and the proof system specifies how these assertions can be composed in a syntax-directed *a la* Hoare deduction of abstract non-interference.
The paper is structured as follows. In Sect. 2, we provide the necessary formal background in abstract interpretation and in program semantics, explaining the

notation that will be used along the paper. In Sect. 3, we recall the recent generalisation [15] of abstract non-interference, which extends the notion introduced in language-based security [12] to any field of computer science where we are interested in understanding the degree of interference between two different groups of data. Sect. 4, is the core of the paper, here we describe the proof system. In particular, by means of some examples, we explain the restrictions that we have to consider for abstract non-interference in order to being able to characterise a proof system inductive on the syntactic structure of simple imperative programs. This section is split in several parts. In Sect 4.1 we introduce a system for the derivation of assertions about invariant properties. A property is invariant if it is left unchanged by the execution of a program. This kind of properties are important for the characterisation of abstract non-interference properties when dealing with loops. In Sect. 4.2, we describe a sound proof system for abstract non-interference in the most general context possible, while in Sect.4.3, we show how we can make the system complete, losing in this case, the effectiveness of the system. Finally, in Sect. 4.4, we show how we can extend the system to non-deterministic paradigms.
In Sect. 5, we instantiate the proof system for abstract non-interference to the particular context of language-based security. This specialisation allows us also to understand why it is not possible to generate a similar system for abstract non-interference where we allow some confidential information to flow, i.e., declassified [12, 20, 3].

This is an extended and revised version of [13].

2 Preliminaries

2.1 Basic notions

Sets are usually denoted with capital letters. If S and T are sets, then $\wp(S)$ denotes the powerset of S, $S \setminus T$ denotes the set-difference between S and T, $S \subset T$ denotes strict inclusion, and for a function $f : S \to T$ and $X \subseteq S$, $f(X) \stackrel{\text{def}}{=} \{f(x) \mid x \in X\}$. We will often denote $f(\{x\})$ as $f(x)$. By $g \circ f$ we denote the composition of the functions f and g, i.e., $g \circ f \stackrel{\text{def}}{=} \lambda x.g(f(x))$. $id \stackrel{\text{def}}{=} \lambda x.\, x$. $\langle P, \leq\rangle$ denotes a poset P with ordering relation $\leq$, while $\langle P, \leq, \vee, \wedge, \top, \bot\rangle$ denotes a complete lattice P, with ordering $\leq$, *lub* $\vee$, *glb* $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. $S \longrightarrow T$ denotes the set of all functions from S to T. We use the symbol $\sqsubseteq$ to denote point-wise ordering between functions: If S is any set, P a poset, and $f, g : S \to P$ then $f \sqsubseteq g$ if for all $x \in S$, $f(x) \leq_P g(x)$. Let C and A be complete lattices, then, $C \stackrel{\text{m}}{\longrightarrow} A$ and $C \stackrel{\text{c}}{\longrightarrow} A$, denote, respectively, the set of all monotone and (Scott-)continuous functions from C to A. Recall that $f \in C \stackrel{\text{c}}{\longrightarrow} A$ iff f preserves *lub*'s of (nonempty) chains iff f preserves *lub*'s of directed subsets, and $f : C \to A$ is (completely) additive if f preserves *lub*'s of all subsets of C (empty set included).

2.2 Abstract interpretation basics

In the following of this paper we will use the standard framework of abstract interpretation [8, 9] for modelling the semantic observations of program semantics. The idea is that, instead of observing the concrete semantics of programs, namely the concrete values of observable data, the analysers can only observe *properties* of these data, namely *abstract semantics* of the program. In other words, abstract interpretation is used for reasoning on *properties* rather than reasoning on data values. For example, instead of computing on integers we might compute on more abstract properties, such as the sign $\{0+, 0-, 0\}$ or parity $\{\mathtt{ev}, \mathtt{od}\}$. Consider the program $\mathtt{sum}(x, y) = x + y$, then it is abstractly interpreted as $\mathtt{sum}^*$: $\mathtt{sum}^*(0+, 0+) = 0+$, $\mathtt{sum}^*(0-, 0-) = 0-$, but $\mathtt{sum}^*(0+, 0-) =$ *"I don't know"* since we are not able to determine the sign of the sum of a negative number with a positive one (modelled by the fact that the result can be any value). Analogously, $\mathtt{sum}^*(\mathtt{ev}, \mathtt{ev}) = \mathtt{ev}$, $\mathtt{sum}^*(\mathtt{od}, \mathtt{od}) = \mathtt{ev}$ and $\mathtt{sum}^*(\mathtt{ev}, \mathtt{od}) = \mathtt{od}$.
Abstract interpretation is highly developed theory where abstract domains can be equivalently formulated either in terms of Galois connections or closure operators [9]. More formally, given a concrete domain C we choose to describe abstractions of C as upper closure operators.

Definition 2.1 *An* upper closure operator *(uco for short)* $\rho : C \to C$ *on a poset* C *is monotone, idempotent, and extensive:* $\forall x \in C.\ x \leq_C \rho(x)$.

The upper closure operator is the function that maps the concrete values with their abstract properties, namely with the best possible approximation of the concrete value in the abstract domain. For example, the operator used in the introduction $\mathit{Sign} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, on the powerset of integers, associates each set of integers with its sign: $\mathit{Sign}(\varnothing) =$ *"none"*, $\mathit{Sign}(S) = 0+$ if $\forall n \in S.\ x \geq 0$, $\mathit{Sign}(0) = 0$, $\mathit{Sign}(S) = 0-$ if $\forall n \in S.\ n \leq 0$ and $\mathit{Sign}(S) =$ *"I don't know"* otherwise. The used property names *"none"*, 0+,0,0− and *"I don't know"* are the names of the following sets in $\wp(\mathbb{Z})$: $\varnothing$, $\left\{\ n \in \mathbb{Z} \,\middle|\, n \geq 0\ \right\}$, $\{0\}$, $\left\{\ n \in \mathbb{Z} \,\middle|\, n \leq 0\ \right\}$ and $\mathbb{Z}$.
Analogously, the operator $\mathit{Par} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ associates each set of integers with its parity, $\mathit{Par}(\varnothing) =$ *"none"* $= \varnothing$, $\mathit{Par}(S) = \mathtt{ev} = \left\{\ n \in \mathbb{Z} \,\middle|\, n \text{ is even}\ \right\}$ if $\forall n \in S.\ n$ is even, $\mathit{Par}(S) = \mathtt{od} = \left\{\ n \in \mathbb{Z} \,\middle|\, n \text{ is odd}\ \right\}$ if $\forall n \in S.\ n$ is odd and $\mathit{Par}(S) =$ *"I don't know"* $= \mathbb{Z}$ otherwise. Namely the abstract elements, in general, correspond to the set of values with the property they represent. Formally, closure operators ρ are uniquely determined by the set of their fix-points $\rho(C)$ and in the following we will often use this representation. Hence, we can describe the two domains *Sign* and *Par* simply by providing the set of fix-points of the corresponding closures: $\mathit{Sign} = \{\mathbb{Z}, 0+, 0, 0-, \varnothing\}$ and $\mathit{Par} = \{\mathbb{Z}, \mathtt{ev}, \mathtt{od}, \varnothing\}$. In Fig. 1 we have a graphical representation of the sublattices of $\wp(\mathbb{Z})$ corresponding to the abstract domains *Sign* and *Par*.

For upper closures, $X \subseteq C$ is the set of fix-points of $\rho \in \mathit{uco}(C)$ iff X is a *Moore-family* of C, i.e., $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$ — where $\wedge\varnothing =$

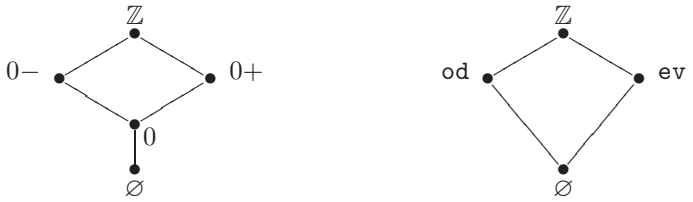

Figure 1: The *Sign* and *Par* domains.

$\top \in \mathcal{M}(X)$. The set of all upper closure operators on C, denoted $uco(C)$, is isomorphic to the so called *lattice of abstract interpretations of* C [9]. If $\langle C, \leq_C, \wedge_C, \vee_C, \top, \bot\rangle$ is a complete lattice then $uco(C)$ ordered point-wise is also a complete lattice, $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \mathbb{T}, id\rangle$ where for every $\rho, \eta \in uco(C)$, $\{\rho_i\}_{i\in I} \subseteq uco(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\forall y \in C.\ \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$; $(\sqcap_{i\in I}\rho_i)(x) = \wedge_{i\in I}\rho_i(x)$; and $(\sqcup_{i\in I}\rho_i)(x) = x \Leftrightarrow \forall i \in I.\ \rho_i(x) = x$, and $\mathbb{T} = \lambda x.\top$ is the closure mapping all concrete elements to the lattice top while id is the identity closure mapping each elements to itself. The *disjunctive completion* of an abstract domain $\rho \in uco(C)$ is the most abstract domain able to represent the concrete disjunction of its objects: $\curlyvee(\rho) = \bigsqcup\{\eta \in uco(C) | \eta \sqsubseteq \rho$ and η is additive$\}$. ρ is disjunctive iff $\curlyvee(\rho) = \rho$ (cf. [9]). Closure operators and partitions are related concepts. If π is a partition (viz. an equivalence relation), then $[\cdot]_\pi$ is the corresponding equivalence class. A closure $\eta \in uco(\wp(S))$ induces a partition on S: $\{\ [x]_\eta \mid\ x \in S\ \}$, where $[x]_\eta \stackrel{\text{def}}{=} \{\ y \mid\ \eta(x) = \eta(y)\ \}$. The most concrete closure that induces the same partition of values as η is $\Pi(\eta) \stackrel{\text{def}}{=} \curlyvee(\{\ [x]_\eta \mid\ x \in S\ \})$. η is *partitioning* if $\eta = \Pi(\eta)$ [22]. The idea is that $\Pi(\eta)$ is the most concrete closure such that for any $y \in \Pi(\eta(x))$: $\Pi(\eta(x)) = \Pi(\eta(y))$, while in general $\eta(y) \subseteq \eta(x)$.

2.3 The imperative language

In this section, we introduce the syntax of a simple programming language, Imp [28], which is a small language of while programs.

Syntax: The deterministic fragment. First of all, we list the syntactic sets associated with Imp: Values $\mathbb{V}$; Truth values $\mathbb{B} = \{true, false\}$; Variables *Var*; Arithmetic expression *Aexp*; Boolean expression *Bexp*; Commands *Com*. We assume that the syntactic structure of numbers is given. We will use the following convention: m, n range over values $\mathbb{V}$; x, y range over variables *Var*; a ranges over arithmetic expression *Aexp*; b ranges over boolean expression *Bexp*; c ranges over commands *Com*. We describe the arithmetic and boolean expressions in *Aexp Bexp* as follows:

$$\begin{array}{lcl} a & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \cdot a_1 \\ b & ::= & true \mid false \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \end{array}$$

Finally, for commands we have the following abstract syntax:

$$c ::= \textbf{skip} \mid x := a \mid c_0; c_1 \mid \textbf{while } b \textbf{ do } c \textbf{ endw} \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$$

Note that

$$\begin{array}{rl} \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \equiv & \textbf{while } b \textbf{ do } c_0; b := \mathit{false} \ \textbf{ endw}; \\ & \textbf{while } \neg b \textbf{ do } c_1; b := \mathit{true} \ \textbf{ endw} \end{array}$$

Therefore, in the following we will consider the language Imp, omitting the control statement **if**.

Semantics. As usual the set of values $\mathbb{V}$ can be structured as a flat domain with additional bottom element $\perp$, denoting the value of not initialized variables. In the following, we will denote by $\mathit{Var}(P)$ the set of variables of the program $P \in$ Imp. We consider the well-known (small-step) operational semantics of Imp, $\longrightarrow$, in Table 1 [28]. Here $[\![e]\!](s) = n$ denotes that standard evaluation of the arithmetic expression e, in the state s, is the value $n \in \mathbb{V}$ and $[n \mapsto x]$ denotes that the value n is substituted to each occurrence of the variable x.
The operational semantics naturally induces a transition relation on a set of states Σ, denoted $\rightarrow$, specifying the relation between a state and its possible successors. $\langle \Sigma, \rightarrow \rangle$ is a transition system. In this transition system, states are representations of the memory, i.e., associations between variables and values. For this reason, in the following we will denote states as tuples of values, the values associated with the variables by the given state. Therefore, if $|\mathit{Var}(P)| = n$, then Σ is the set of all n-tuples of values, i.e., $\Sigma = \mathbb{V}^n$. We abuse notation by denoting with $\perp$ the state where all variables are undefined. Note that, in the rules provided in Table 1 we have transitions between *configurations*, i.e., pairs $\langle \text{code}, \text{state} \rangle$. In the following by $[\![P]\!]$ we will denote the denotational semantics of the program P.

We follow Cousot's construction [7], by defining semantics, at different levels of abstractions, as the abstract interpretation of the maximal trace semantics of a transition system associated with each well-formed program. In the following, Σ^+ and $\Sigma^\omega \stackrel{\text{def}}{=} \mathbb{N} \longrightarrow \Sigma$ denote respectively the set of finite nonempty and infinite sequences of symbols in Σ. Given a sequence $\sigma \in \Sigma^\infty \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$, its length is denoted $|\sigma| \in \mathbb{N} \cup \{\omega\}$ and its i-th element is denoted σ_i. A non-empty finite (infinite) *trace* $\sigma \in \Sigma^\infty$ is a finite (infinite) sequence of program states where two consecutive elements are in the transition relation $\rightarrow$, i.e., for all $i < |\sigma|$: $\sigma_i \rightarrow \sigma_{i+1}$. For each $\sigma \in \Sigma^\infty$ $\sigma_\vdash$ denotes initial state of σ, and for each $\sigma \in \Sigma^+$ $\sigma_\dashv$ denotes its final state. The *maximal trace semantics* [7] of a transition system associated with a program P is $\langle\!|P|\!\rangle \stackrel{\text{def}}{=} \langle\!|P|\!\rangle^+ \cup \langle\!|P|\!\rangle^\omega$, where $\Sigma_\dashv \subseteq \Sigma$ is a set of final/blocking states and $\Sigma_\vdash$ denotes the set of initial states for P. Then $\langle\!|P|\!\rangle^\omega = \{\sigma \in \Sigma^\omega \mid \forall i \in \mathbb{N} \,.\, \sigma_i \rightarrow \sigma_{i+1}\}$, $\langle\!|P|\!\rangle^+ = \{\sigma \in \Sigma^+ \mid \sigma_\dashv \in \Sigma_\dashv, \ \forall i \in [1, |\sigma|) .\, \sigma_{i-1} \rightarrow \sigma_i\}$.
The denotational semantics is obtained by abstracting trace semantics to the input and output states only, i.e.,

$$[\![P]\!] = \lambda \sigma_\vdash . \begin{cases} \sigma_\dashv & \text{if } \sigma \in \langle\!|P|\!\rangle^+ \\ \perp & \text{if } \sigma \in \langle\!|P|\!\rangle^\omega \end{cases}$$

$$\langle \mathbf{nil}, s\rangle \qquad \frac{[\![e]\!](s) = n \in \mathbb{V}}{\langle x := e, s\rangle \longrightarrow \langle \mathbf{nil}, s[n \mapsto x]\rangle}$$

$$\frac{\langle c_0, s\rangle \longrightarrow \langle c_0', s'\rangle}{\langle c_0; c_1, s\rangle \longrightarrow \langle c_0'; c_1, s'\rangle} \qquad \frac{\langle c_1, s_0\rangle \longrightarrow \langle c_1', s_0'\rangle}{\langle \mathbf{nil}; c_1, s\rangle \longrightarrow \langle c_1', s_0'\rangle}$$

$$\frac{\langle b, s\rangle \longrightarrow \mathit{true},\ \langle c, s\rangle \longrightarrow \langle c', s'\rangle}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{endw}, s\rangle \longrightarrow \langle c'; \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{endw}, s'\rangle}$$

$$\frac{\langle b, s\rangle \longrightarrow \mathit{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{endw}, s\rangle \longrightarrow \langle \mathbf{nil}, s\rangle}$$

Table 1: Operational semantics of Imp

In this case, it is assumed that the output observation of an infinite computation is the state where all variables are undefined [7].

The non-deterministic fragment. A simple way to introduce some basic issues in order to obtain non-deterministic languages is to extend the simple imperative language Imp by an operation of non-deterministic choice. We define in this way the language Nd-Imp, whose commands are defined in the following way:

$$c ::= \mathbf{skip} \mid x := a \mid c_0; c_1 \mid \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{endw} \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid c_0 \,\square\, c_1$$

We have to extend the operational semantics with the rules for the non-deterministic choice:

$$\frac{\langle c_0, s\rangle \longrightarrow \langle c_0', s'\rangle}{\langle c_0 \,\square\, c_1, s\rangle \longrightarrow \langle c_0', s'\rangle} \qquad \frac{\langle c_1, s\rangle \longrightarrow \langle c_1', s'\rangle}{\langle c_0 \,\square\, c_1, s\rangle \longrightarrow \langle c_1', s'\rangle}$$

We consider now the possibilistic extension of denotational semantics for non-deterministic systems, and we abuse notation by denoting this semantics by

$$[\![P]\!] = \lambda s.\ \{\ \sigma_{\dashv} \mid\ \sigma \in \langle\!|P|\!\rangle^{+},\ \sigma_{\vdash} = s\ \} \cup \{\ \bot \mid\ \exists \sigma \in \langle\!|P|\!\rangle^{\omega}.\, \sigma_{\vdash} = s\ \}$$

3 Abstract Non-Interference

In this section, we introduce the notion of abstract non-interference [12, 15], i.e., a weakening of non-interference given in terms of observers modelled by means of abstract interpretations of concrete semantics. We will start from standard notion on non-interference (NI for short), originally introduced in language-based security [6, 17, 23], and here generalized to any kind of classification of

data (intended as program variables), where we are interested in understanding if a given class of data interferes with another class of data. In other words, we generalize the public/private data classification in language-based security to a generic observable/internal classification.
Consider variables distinguished into two classes, *internal* (denoted $*$) and *observable* (denoted $\circ$). The internal data correspond to those variables that have not to interfere with the observable ones. It is clear that, this classification is static, in the sense that it is fixed and cannot change dynamically, but it is not a property of data involved as in the security context. It simply characterises the NI policy we have to verify/define.
Both the input and the output variables are partitioned in this way, and the two partitions need not coincide. Hence, if $\mathcal{I}$ denotes the set of input variables and $\mathcal{O}$ denotes the set of output variables, we have four classes of data: $\mathcal{I}_*$ which are the input internal variables, $\mathcal{I}_\circ$ which are the input observable variables, $\mathcal{O}_*$ which are the output internal variables, that cannot be observed, and $\mathcal{O}_\circ$ which are the output observable variables. Note that the formal distinction between $\mathcal{I}$ and $\mathcal{O}$ is used only to underline that there can be different partitions of input and output, namely $\mathcal{I}_* = \mathcal{O}_*$ need not hold. In general, we have the same set of variables in input and in output, hence $\mathcal{I} = \mathcal{O} = \mathit{Var}(P)$.
Informally, non interference can be reformulated by saying that if we fix the values of variables in $\mathcal{I}_\circ$ and we let values of variables in $\mathcal{I}_*$ change, we have not to observe any difference in the values of variables in $\mathcal{O}_\circ$. Indeed if this happens it means that $\mathcal{I}_*$ interferes with $\mathcal{O}_\circ$. We will use the following notation: $\mathbb{I}_* \stackrel{\text{def}}{=} \mathbb{V}^{|\mathcal{I}_*|}$, $\mathbb{O}_* \stackrel{\text{def}}{=} \mathbb{V}^{|\mathcal{O}_*|}$, $\mathbb{I}_\circ \stackrel{\text{def}}{=} \mathbb{V}^{|\mathcal{I}_\circ|}$ and $\mathbb{O}_\circ \stackrel{\text{def}}{=} \mathbb{V}^{|\mathcal{O}_\circ|}$, where $|X|$ denotes the cardinality of the set of variables X. Consider $\mathbb{C} \in \{\mathbb{I}_*, \mathbb{I}_\circ, \mathbb{O}_*, \mathbb{O}_\circ\}$, in the following, we abuse notation by denoting $v \in \mathbb{C}$ the fact that v is a possible tuple of values for the vector of variables evaluated in $\mathbb{C}$, e.g., $v \in \mathbb{I}_*$ is a vector of values for the variables in $\mathcal{I}_*$. Moreover, if x is a tuple of variables in $\mathcal{O}$ (analogous for $\mathcal{I}$) we denote as x^* $[x^\circ]$ the projection of tuple of variables x only on the variables in $\mathcal{O}_*$ $[\mathcal{O}_\circ]$ (analogous when we consider values instead of variables). In the following, when a variable is internal [observable] we will also use the notation $x : *$ $[x : \circ]$. At this point, we can reformulate standard non-interference, wrt a fixed partition of variables $\mathcal{P} \stackrel{\text{def}}{=} \{\mathcal{I}_\circ, \mathcal{I}_*, \mathcal{O}_\circ, \mathcal{O}_*\}$:

$$\boxed{\begin{array}{c}\text{A program } P\text{, satisfies } \textit{non-interference} \text{ between } * \text{ and } \circ \text{ if} \\ \forall v \in \mathbb{I}_\circ, \forall v_1, v_2 \in \mathbb{I}_* \,.\, (\llbracket P \rrbracket(v_1, v))^\circ = (\llbracket P \rrbracket(v_2, v))^\circ\end{array}} \quad (1)$$

3.1 Weakening Non-Interference

Consider the program $P \stackrel{\text{def}}{=} x := |x| * \mathit{Sign}(y)$ (seen in the introduction, where $\mathcal{I}_* = \{y\}$ and $\mathcal{I}_\circ = \mathcal{O}_\circ = \{x\}$), suppose that $|\cdot|$ is the absolute value function and suppose $\mathit{Sign}(y)$ returns the sign of y, then "*only a portion of x is*

affected, in this case x's sign. Imagine if an observer could only observe x's absolute value and not x's sign" [6] then we could say that in the program there is non-interference between $*$ and $\circ$. Abstract interpretation provides the most appropriate framework to further develop Cohen's intuition. The basic idea is that an observer can analyze only some properties, modeled as abstract interpretations of the concrete program semantics.
In the following, the *observer* is a pair of abstractions $\langle \eta, \rho \rangle$, with η abstraction on the whole input (internal and observable), $\eta \in \mathit{uco}(\wp(\mathbb{I}))$, ρ abstraction on the observable output, $\rho \in \mathit{uco}(\wp(\mathbb{O}_\circ))$, representing what can be observed about, respectively, the input and output of a program. The fact that in output we consider only observable variables while in input we consider all variables, is intuitively due to the fact that non-interference specifies what can be analyzed in output, which clearly corresponds to what is observable of the output, while it says what we can deduce, with our analysis, about the input, and in this case we can derive information about both the internal (in case of interference) and the observable input variables.
Consider another abstraction on the whole input set of variables, $\phi \in \mathit{uco}(\wp(\mathbb{I}))$. It describes a property on the input which represents *when*, i.e., for which inputs, we are interested in testing non-interference properties. We can say that the idea of abstract non-interference is that a program P satisfies abstract non-interference relatively to a pair of observations η and ρ, and to a property ϕ, denoted $[\phi \vdash (\eta)P(\rho)]$, if, whenever the input values have the same property ϕ then the best correct approximation of the semantics of P, wrt η in input and ρ in output, does not change. This captures precisely the intuition that ϕ-indistinguishable input values provide η, ρ-indistinguishable results, for this reason it can still be considered a non-interference policy. The following definition introduces the notion of abstract non-interference as a generalization of the standard one. In the following, we will often consider $\rho \in \mathit{uco}(\mathbb{I})$ (analogous for $\mathbb{O}$) such that ρ abstracts internal and observable variables in an attribute independent way. Namely, ρ can be split in two independent abstractions, one for the variables in $\mathcal{I}_*$, denoted ρ^*, and one for the variables in $\mathcal{I}_\circ$, denoted ρ°, and we write $\rho = \rho^* \times \rho^\circ$. In general, we can always consider the projection of ρ on the variables in $\mathcal{I}_\circ$ (analogous for $\mathcal{I}_*$): $(\rho)^\circ \stackrel{\text{def}}{=} \lambda \langle y, x \rangle \in \mathbb{I}_* \times \mathbb{I}_\circ .\ \{\ x' \mid \langle y', x' \rangle \in \rho(y, x)\ \}$. It is worth noting that, if ρ is not relational[1] then $\rho^\circ = (\rho)^\circ$.

Definition 3.1 [Abstract non-interference]
Let $\phi, \eta \in \mathit{uco}(\wp(\mathbb{I})), \rho \in \mathit{uco}(\wp(\mathbb{O}_\circ))$.

A program P satisfies $[\phi \vdash (\eta)P(\rho)]$ if
$\forall x_1, x_2 \in \mathbb{I} .\ \phi(x_1) = \phi(x_2) \ \Rightarrow\ \rho((\llbracket P \rrbracket(\eta(x_1)))^\circ) = \rho((\llbracket P \rrbracket(\eta(x_2)))^\circ)$

For instance, in Eq. 1 we have $\phi = \mathbb{T}^* \times \mathit{id}^\circ$, $\eta = \mathit{id}$ and $\rho = \mathit{id}$, where we recall that $\mathbb{T}^* = \lambda x \in \mathbb{I}_* .\ \top$ and $\mathit{id}^\circ = \lambda x \in \mathbb{I}_\circ .\ x$. In the following, we define closures

[1] Here, by relational, we mean not attribute independent, namely a property describing relations of elements, for example $\rho(\langle x, y \rangle) = 0+$ if $x + y \geq 0$ is a relational property.

on $\mathbb{V}^n$ by using closures on $\mathbb{V}$. In this case we abuse notation by supposing that $\rho(\langle x, y\rangle) = \langle \rho(x), \rho(y)\rangle$.

Example 3.2 *Consider the property Sign and Par represented in Fig. 1, Consider $\mathcal{I}_\circ = \{x\}$, $\mathcal{I}_* = \{y\}$ and $\mathbb{I} = \mathbb{Z}$. Let $\phi = Sign$, $\eta = id$, $\rho = Par$, and consider the program fragment:*

$$P \stackrel{def}{=} x := 2 * x * y^2;$$

In the standard notion of non-interference there is a flow of information from variable y to variable x, since x depends on the value of y, i.e., the statement does not satisfy non-interference.
Let us consider $[Sign \vdash (id)P(Par)]$. If $Sign(\langle x, y\rangle) = \langle Sign(x), Sign(y)\rangle = \langle 0+, 0+\rangle$, then the possible outputs are always in `ev`*, indeed the result is always even because there is a multiplication by 2. The same holds if $Sign(\langle x, y\rangle) = \langle 0-, 0-\rangle$. Therefore any possible output value, with a fixed observable input, has the same observable abstraction in Par, which is* `ev`*. Hence $[Sign \vdash (id)P(Par)]$ holds.*

3.2 Basic properties of ANI and blind kernels

Abstract non-interference is parametric on program properties specified as closure operators. We can observe that the property where we cannot observe anything in output, i.e., $[\phi \vdash (\eta)P(\mathbb{T})]$ always holds. Indeed, if a closure identifies some objects, then every more abstract closure will identify at least the same objects. From these simple observations we derive the following basic properties of abstract non-interference.

Proposition 3.3 *[12, 15] Let $\{\phi_i\}_{i\in I}$, $\{\rho_i\}_{i\in I}$, with $I \subset \mathbb{N}$, and let $\phi, \phi_i, \eta \in uco(\wp(\mathbb{I}))$, $\rho, \rho_i \in uco(\wp(\mathbb{O}_\circ))$ and the program $P \in$ Imp.*

1. $[\phi \vdash (\eta)P(\rho)] \;\Leftrightarrow\; \forall \rho_1 \sqsupseteq \rho \,.\, [\phi \vdash (\eta)P(\rho_1)]$;
2. $\forall i.\, [\phi \vdash (\eta)P(\rho_i)] \;\Rightarrow\; [\phi \vdash (\eta)P(\bigsqcap_i \rho_i)]$;
3. $[\phi \vdash (\eta)P(\rho)] \;\Leftrightarrow\; \forall \phi_1 \sqsubseteq \phi \,.\, [\phi_1 \vdash (\eta)P(\rho)]$;
4. $(\forall i.\, [\phi_i \vdash (\eta)P(\rho)] \;\Rightarrow\; [\bigsqcup_i \phi_i \vdash (\eta)P(\rho)])$ *iff* $\forall i.\, \phi_i$ *are partitioning.*

Let us recall that there exists a systematic method for deriving output blind observers from programs by abstract interpretation [12, 15]. This is useful both in automatic program certification for deriving basic assertions, and in order to classify programs in terms of the properties that make non-interference hold. Since (output) observers are characterized by abstract domains, the idea is to define an abstract domain transformer, depending on the program to be analyzed, which is able to transform any abstraction ρ, able to observe interference, into the closest abstraction unable to observe any interference, i.e., *blind*. In this way we can characterize the most powerful blind observer for a given program.

The soundness of this idea is provided by Proposition 3.3(1). In particular, consider a program P and an ANI property $[\phi \vdash (\eta)P(\rho)]$, we know by Proposition 3.3(2) that the most concrete $\rho_1 \sqsupseteq \rho$ such that $[\phi \vdash (\eta)P(\rho_1)]$ always exists unique. We call this domain the *blind kernel of ρ for P* and we denote it with the following notation: $[\phi \vdash (\eta)[\![P]\!](\rho)]$.

4 A proof system for Abstract Non-Interference

In the previous section, we recall that abstract non-interference can be defined in a general framework, where the output variables are not necessarily partitioned into internal and observable data. In other words, we consider the notion of abstract non-interference (ANI for short) $[\phi \vdash (\eta)P(\rho)]$, where the input abstractions ϕ and η are on the whole input domain, while the output property can only abstract the observable data.

Note that the aim of this work is that of providing a proof system, inductive on the syntactic structure of programs, which allows us to deduce ANI properties of a program by combining ANI properties of its syntactic components. In order to obtain this, we have first to understand how we can combine ANI properties of programs depending on the syntactic structure. Before introducing the proof system we have to pay attention to the while statement. In fact, the while has the problem of opening implicit channels of information. We recall that an implicit channel is due to the dependency existing between the variables in the guard of the statement and those modified inside its body, e.g., if we consider $x := 0; \textbf{while } y \textbf{ do } x := 1; y := y - 1$ we have an implicit flow of information from y to x, since the final value of x depends on the initial value of y. These kinds of flows may violate non-interference, hence the rule for the **while** has to avoid them. Hence, we have not to distinguish, from the observation of the output, how many times (zero or more) the **while** body has been executed. This means that the input/output observable property has to be an *invariant* of the loop, namely the execution of the body has to leave this property unchanged. In the following, we describe the *observable invariant proof system.*
At this point, in order to generate the proof system, some restrictions has to be taken into account. One of these restrictions is relevant also for the invariant proof system and for this reason we introduce it here. In particular, in order to handle correctly the assignment, where the value of only one variable is modified, we have to consider only attribute independent abstractions, i.e., abstractions such that the property of a tuple is a tuple of properties, one for each element of the tuple. Hence, when $\rho(\langle x, y\rangle)$ then we have that $\rho = \langle \rho_x, \rho_y\rangle$ namely $\rho(\langle x, y\rangle) = \langle \rho_x(x), \rho_y(y)\rangle$.

4.1 Proof system for observable invariants

In order to derive a proof system for non-interference, when implicit flows may occur, we need to model the properties that are invariant during the execution of programs. Intuitively, an abstraction ρ is invariant for a program fragment

$$\textbf{I1: } c \vdash_{\mathtt{I}} \mathbb{T} \qquad \textbf{I2: } \textbf{skip} \vdash_{\mathtt{I}} \rho \qquad \textbf{I3: } \frac{x : *}{x := e \vdash_{\mathtt{I}} \rho} \qquad \textbf{I4: } \frac{\langle e, x\rangle \vdash_{\mathtt{I}} \rho,\ x : \circ}{x := e \vdash_{\mathtt{I}} \rho}$$

$$\textbf{I5: } \frac{c_1 \vdash_{\mathtt{I}} \rho,\ c_2 \vdash_{\mathtt{I}} \rho}{c_1; c_2 \vdash_{\mathtt{I}} \rho} \qquad \textbf{I6: } \frac{c \vdash_{\mathtt{I}} \rho}{\textbf{while } x > 0 \textbf{ do } c \textbf{ endw} \vdash_{\mathtt{I}} \rho} \qquad \textbf{I7: } \frac{c \vdash_{\mathtt{I}} \rho_1,\ \rho_1 \sqsubseteq \rho}{c \vdash_{\mathtt{I}} \rho}$$

Table 2: Derivation of public invariants of programs.

P, written $P \vdash_{\mathtt{I}} \rho$, when by observing the property ρ of observable inputs, we are not able to observe any difference in the ρ property of the corresponding outputs. In other words, $P \vdash_{\mathtt{I}} \rho$ means that P is observably equivalent to **skip** as regards the observable property ρ. This information is essential in order to certify the lack of implicit flows relatively to an observation. These invariant abstractions are obtained with an *a la* Hoare proof system, where assertions are invariant properties of the form $P \vdash_{\mathtt{I}} \rho$, with $\rho \in \mathit{uco}(\wp(\mathbb{O}_\circ))$. Invariants of expressions are parametric on a variable, the observable variable to which they can be assigned. In the following, being the closure ρ a tuple of closures on the single variables, for each observable variable x, we denote by ρ_x the component of ρ applied to x.

Definition 4.1 *Given an expression e in* Imp *and a variable x, we say that the property ρ of the variable x is invariant in e, and we write $\langle e, x\rangle \vdash_{\mathtt{I}} \rho$, if:*

$$\forall v \in \mathbb{I}\,.\,\rho_x([\![e]\!](v)) = \rho_x(v_{|_x})$$

where for any expression e, $[\![e]\!] : \Sigma \longrightarrow \mathbb{V}$ is the standard semantics of expressions and where $v_{|_x}$ is the value for x in the tuple v.

The intuition is that e does not change the property ρ of the value of x inside v. We extend this definition of invariant properties of expressions in order to define invariant properties of statements/programs.

Definition 4.2 *Given a program P in the language* Imp*, we say that a property ρ on observable outputs is an invariant in the program P, denoted $P \vdash_{\mathtt{I}} \rho$, if*

$$\forall v \in \mathbb{I}\,.\,\rho(([\![P]\!](v))^\circ) = \rho(v^\circ)$$

Observable invariants for programs can be derived by induction on the syntax of Imp by using the proof system $\mathtt{I} = \{\textbf{I1}, \ldots, \textbf{I7}\}$ whose rules are defined in Table 2 and explained in the following.

- Rule **I1** says that the property $\mathbb{T} = \lambda x \in \mathbb{O}_\circ.\ \top$ is invariant for any program. This holds since $\mathbb{T}$ is the property unable to distinguish any difference among observable values. Therefore, any change due to the execution of a program cannot be observed through the property $\mathbb{T}$.

- Rule **I2** says that any property is invariant for the program **skip**. This holds since **skip** does not change observable data, and therefore observable data properties are left unchanged.

- When we have an assignment to internal variables, then the semantics behaves like **skip** relatively to observable values, therefore rule **I3** is similar to **I2**, since, by definition, invariants are defined only for observable variables.

- In **I4**, if a property is invariant for the evaluation of an expression as regards the observable variable x, then it is invariant for the assignment of the expression to x. Consider, for example, the expression $x+2$, then the property *Sign* (Fig. 1) is not invariant, since if we consider the input value $x = -1$, then we have that $\mathit{Sign}(x+2) = \mathit{Sign}(1) = + \neq \mathit{Sign}(x) = -$. On the other hand, we have that *Par* (Fig. 1) is invariant for this expression as regards the variable x, since the operation $x+2$ doesn't change the parity of the value assigned to x. At this point if the statement is $x := x + 2$, then we have that $x := x + 2 \vdash_{\mathtt{I}} \mathit{Par}$.

- Rule **I5** says that the invariants distribute on the sequential composition. Hence, if for example we have the program $x := x + 2;\ y := y - 1$, where x is observable ($x : \circ$) and y is internal ($y : *$), then we know, by **I3**, that $y := y - 1 \vdash_{\mathtt{I}} \mathit{Par}$ and by **I4** that $x := x + 2 \vdash_{\mathtt{I}} \mathit{Par}$. Therefore, we obtain $x := x + 2;\ y := y - 1 \vdash_{\mathtt{I}} \mathit{Par}$.

- Rule **I6** states that, given a **while**-statement, if a property is invariant for the body, then the same property is invariant for the whole statement. This rule holds since the only modifications of variables made by the **while**, are made by its body.

- Weakening (**I7**) says that any more abstract property of an invariant is still invariant.

A derivation in the proof system of observable invariants in Table 2 is denoted $\vdash_{\mathtt{DI}}$. The following theorem shows that the proof system for invariants is *sound* as regards the given definition of invariant properties (Def. 4.2).

Theorem 4.3 *Let $P \in$ Imp and $\rho \in uco(\mathbb{O}_\circ)$, such that $\rho = \langle \rho_1, \ldots, \rho_n \rangle$, where $n = \left|\left\{\ x \in \mathit{Var}(P) \;\middle|\; x : \circ\ \right\}\right|$. If $\vdash_{\mathtt{DI}} (P \vdash_{\mathtt{I}} \rho)$ then $P \vdash_{\mathtt{I}} \rho$ holds.*

Proof. The proof is by induction on the rules in Table 2. The rules **I1** and **I2** are trivial since the closure $\mathbb{T}$ makes each element equal to the element $\top$, while $(([\![\mathbf{skip}]\!](x))^\circ = x^\circ$ by definition of **skip** and therefore for each ρ we have $\rho((([\![\mathbf{skip}]\!](x))^\circ) = \rho(x^\circ)$. As far as **I3** is concerned, the assignment semantics guarantees that $y : *$ implies $\rho((([\![x := e]\!](v))^\circ) = \rho(v^\circ)$, being v° left unchanged by the assignment. Consider **I4**, then the hypothesis $\langle e, x \rangle \vdash_{\mathtt{I}} \rho$ says that $\forall v : \mathbb{O}\ .\ \rho_x(v_{|_x}) = \rho_x([\![e]\!](v))$. We have to prove that, with this hypothesis, $\rho((([\![x := e]\!](v))^\circ) = \rho(v^\circ)$ holds. Since we have that the

abstraction ρ of a tuple of values is a tuple of abstractions, then after the execution of $x := e$ we obtain $\rho(v[x \mapsto [\![e]\!](v)]) = \rho(v^\circ)[x \mapsto \rho_x(v_{|_x})] = \rho(v^\circ)$. Consider now **I5** and suppose that $c_1 \vdash_{\mathtt{I}} \rho$ and $c_2 \vdash_{\mathtt{I}} \rho$, namely $\forall x \in \mathbb{I}$ we have $\rho(([\![c_1]\!](x))^\circ) = \rho(x^\circ)$ and $\rho(([\![c_2]\!](x))^\circ) = \rho(x^\circ)$. We have the following equalities $\rho(([\![c_1; c_2]\!](x))^\circ) = \rho(([\![c_2]\!]([\![c_1]\!](x)))^\circ) = \rho(([\![c_2]\!](x'))^\circ) = \rho(x'^\circ)$ with $x' = [\![c_1]\!](x)$ therefore $\rho(x'^\circ) = \rho(x^\circ)$, so we have the thesis. **I6** holds since the only modifications that the statement **while** $x > 0$ **do** c **endw** can do are through c. Indeed, if by hypothesis we have $\rho(([\![c]\!](v))^\circ) = \rho(v^\circ)$, then we can prove, by induction on the number of executions of c in the semantics of the **while**, that the rule holds. In fact, the base considers 0 executions of c, namely $[\![\textbf{while } x > 0 \textbf{ do } c \textbf{ endw}]\!] = [\![\textbf{skip}]\!]$, and trivially we have $\rho(([\![\textbf{skip}]\!](v))^\circ) = \rho(v^\circ)$. Suppose now that for n executions of c, the semantics of the while has ρ as observable invariant, namely $\rho(([\![\textbf{while } x > 0 \textbf{ do } c \textbf{ endw}]\!](v))^\circ) = \rho(v^\circ)$, we have to prove that the same holds for c; **while** $x > 0$ **do** c **endw**. But combining together the hypothesis on c and the inductive hypothesis, by Rule **I5**, we have the thesis. Finally, **I7** is straightforward from the definition of invariants. □

Note that the proof system is not complete since Rule **I5** introduces incompleteness. In fact, $x := x + 1; x := x + 1 \vdash_{\mathtt{I}} Par$ holds, while $x := x + 1 \vdash_{\mathtt{I}} Par$ does not hold.

4.2 Proof system for abstract non-interference

We can now introduce a proof system for abstract non-interference. As underlined before, in order to define a proof system, some restrictions have to be taken into account. Let us first understand what happens for sequential composition, namely, suppose we know that $[\phi \vdash (\eta)c_1(\rho)]$ and that $[\phi \vdash (\eta)c_2(\rho)]$, then we wonder what we can say about $c_1; c_2$. It is clear that, if we want to "compose" these ANI properties, the output observation of the first statement has to be the same as the input observation ϕ of the second statement, and this implies that also the input selection observation ϕ has to be defined only on observable data, namely has to be of the kind $\phi = \mathbb{T}^* \times \phi^\circ$, exactly as it happens for the output observation. However, if $\mathcal{O}_\circ = \mathcal{O}$ then both ϕ° and ρ are defined on the whole data domains. In the following, in order to avoid confusion, we will abuse notation by writing $[\phi^\circ \vdash (\eta)P(\rho)]$ instead of $[\mathbb{T}^* \times \phi^\circ \vdash (\eta)P(\rho)]$.
In the previous section, we describe another necessary restriction, i.e., the use of only attribute independent abstractions. For the assignment rule, this condition is only sufficient for making the property hold, but it is necessary in order to rewrite the ANI property of an assignment in terms only of the ANI property of the evaluation of the expression. Otherwise, in fact, the ANI property of the assignment would depend on the whole memory, and not only on the considered expression. For example, consider the simpler case where $\eta = id$ and also $\phi^\circ = id^\circ$, then $[id^\circ \vdash (id)x := e(\rho)]$ requires that, for all memories such that $\sigma^\circ = \sigma_1^\circ$, we have $\rho(([\![x := e]\!](\sigma))^\circ) = \rho(([\![x := e]\!](\sigma_1))^\circ)$, namely we have $\rho(\sigma[x \mapsto [\![e]\!](\sigma)]^\circ) = \rho(\sigma_1[x \mapsto [\![e]\!](\sigma_1)]^\circ)$. Clearly, if ρ is relational, whether this property holds or not depends, not only on the evaluation of e in the different

$$\textbf{R1: } [\phi^\circ \vdash (\eta)c(\mathbb{T})] \qquad \textbf{R2: } \frac{\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)}{[\phi^\circ \vdash (\eta)\mathbf{skip}(\rho)]}$$

$$\textbf{R3: } \frac{x \models [\phi^\circ \vdash (\eta)e(\rho)],\ [\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)],\ x : \circ}{[\phi^\circ \vdash (\eta)x := e(\rho)]} \qquad \textbf{R4: } \frac{x : *,\ \Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)}{[\phi^\circ \vdash (\eta)x := e(\rho)]}$$

$$\textbf{R5: } \frac{c \vdash_{\mathtt{I}} \rho}{[\rho \vdash (id)\mathbf{while}\ x > 0\ \mathbf{do}\ c\ \mathbf{endw}(\rho)]}$$

$$\textbf{R6: } \frac{[\phi^\circ \vdash (\eta)c_1(\rho)],\ [\rho \vdash (id)c_2(\rho_1)],\ [\rho, \rho_1 \text{ additive if } \eta \neq id]}{[\phi^\circ \vdash (\eta)c_1; c_2(\rho_1)]}$$

$$\textbf{R7: } \frac{[\phi^\circ \vdash (id)c(\rho)],\ \Pi(\phi^\circ) \sqsubseteq \Pi((\eta)^\circ),\ \rho \text{ additive}}{[\phi^\circ \vdash (\eta)c(\rho)]} \qquad \textbf{R8: } \frac{[\phi^\circ \vdash (id)c(\rho)]}{[\phi^\circ \vdash (id)c(\curlyvee(\rho))]}$$

$$\textbf{R9: } \frac{[\phi_1^\circ \vdash (\eta)c(\rho_1)],\ \phi^\circ \sqsubseteq \phi_1^\circ,\ \rho_1 \sqsubseteq \rho}{[\phi^\circ \vdash (\eta)c(\rho)]} \qquad \textbf{R10: } \frac{\forall i \in I\,.\,[\phi_i^\circ \vdash (\eta)c(\rho)],\ \phi_i^\circ \text{ partitioning}}{[\bigsqcup_{i\in I} \phi^\circ \vdash (\eta)c(\rho)]}$$

$$\textbf{R11: } \frac{\forall i \in I\,.\,[\phi^\circ \vdash (\eta)c(\rho_i)]}{[\phi^\circ \vdash (\eta)c(\bigsqcap_{i\in I} \rho_i)]} \qquad \textbf{R12: } \frac{\forall i \in I\,.\,[\phi^\circ \vdash (\eta)c(\rho_i)]}{[\phi^\circ \vdash (\eta)c(\bigsqcup_{i\in I} \rho_i)]}$$

Table 3: Axiomatic abstract non-interference

memories, but also on the relation between these evaluations and the rest of the corresponding memories, even if they remain the same.
At this point, we can give sufficient conditions for proving that a statement satisfies abstract non-interference by inductively analyzing its sub-components. The rules of this proof system are specified in Table 3.

- Rule **R1** says that if the output observation is $\mathbb{T} = \lambda x \in \mathbb{O}_\circ.\ \top$, then the input observation can be any. Again, this holds because $\mathbb{T}$ is not able to distinguish different public data.

- Rule **R2** says that **skip** satisfies non-interference for any possible observer such that the partition induced by input selection observation is more concrete than the one induced by the I/O observation of the program semantics. Let us recall that $\Pi(\rho)$ denotes the partition induced by the closure ρ, and that $(\eta)^\circ \stackrel{\text{def}}{=} \lambda\langle y, x\rangle \in \mathbb{I}_* \times \mathbb{I}_\circ.\ \left\{\ x' \ \middle|\ \langle y', x'\rangle \in \eta(y, x)\ \right\}$ (see Sect. 3.1). This condition is necessary since in this case abstract non-interference corresponds to saying $\forall x_1, x_2\,.\,\phi^\circ(x_1^\circ) = \phi^\circ(x_2^\circ)\ \Rightarrow\ \rho((\eta(x_1))^\circ) = \rho((\eta(x_2))^\circ)$ which holds iff $\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)$.

- Rule **R3** considers a notion of non-interference extended to expressions and depending on a fixed variable to which the expression is assigned.

Formally, we can define non-interference for expressions as follows:

$$x \models [\phi^\circ \vdash (\eta)e(\rho)] \text{ iff } \forall v_1, v_2 \in \mathbb{I} \,.\, \phi^\circ(v_1) = \phi^\circ(v_2) \;\Rightarrow \\ \rho_x([\![e]\!](\eta(v_1))) = \rho_x([\![e]\!](\eta(v_2)))$$

We note that the assignment changes only the variable x, all other observable variables (if there are some) are left unchanged. For this reason we need the condition on the partition induced by the involved closures $\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)$ (between square brackets since it is required only when there are more than one observable variable), since for all observable variables different from x the assignment behaves like **skip**.

Example 4.4 *Consider the program fragment*

$$x_1 := 2 * y * x_2$$

with $\mathcal{I}_\circ = \mathcal{O}_\circ = \{x_1, x_2\}$*, then* $[\mathbb{T} \vdash (id)x_1 := 2 * y * x_2(Par)]$ *does not hold since, if the input has the form* $\langle y, x_1, x_2\rangle$

$$Par(([\![x_1 := 2 * y * x_2]\!](y, x_1, 3))^\circ) = \langle \mathtt{ev}, \mathtt{od}\rangle \text{ while} \\ Par(([\![x_1 := 2 * y * x_2]\!](y, x_1', 2))^\circ) = \langle \mathtt{ev}, \mathtt{ev}\rangle.$$

And indeed $\Pi(\mathbb{T}) \not\sqsubseteq \Pi(Par)$*, namely* $\mathbb{T}(3) = \mathbb{T}(2) = \top$ *doesn't imply* $Par(3) = Par(2)$*.*

This condition between the partitions induced by the abstractions is not necessary when the program contains only one observable variable. Consider, for instance $x := y * 2$ ($\mathcal{O}_\circ = \{x\}$), we have that $[\mathbb{T} \vdash (id)y * 2(Par)]$, namely the multiplication by 2 hides the parity property of the computed value. This implies that $[\mathbb{T} \vdash (id)x := y * 2(Par)]$.

- Rule **R4** says that an assignment to an internal variable, from the observable point of view, behaves like **skip**, therefore for this kind of assignments the rule is like **R2**. This means that also in this case abstract non-interference corresponds to saying $\phi^\circ(x^\circ) = \phi^\circ(x'^\circ) \;\Rightarrow\; \rho((\eta(x))^\circ) = \rho((\eta(x'))^\circ)$.

- Rule **R5** controls the **while**-statement when $\eta = id$. In particular $c \vdash_{\mathrm{I}} \rho$ states that the program c is not acting on the property ρ of the observable data, namely ρ is invariant in the execution of c, in the sense that the property ρ of observable data is not changed by the execution of c. If this happens then the behaviour of c observed by means of ρ is the same as the program **skip**, and therefore whether the **while** is executed or not is not distinguishable from an observer.

- Rule **R6** shows how we can compose the non-interference properties in presence of sequential composition of programs. In particular, two programs c_1 and c_2 can be composed when c_1 and c_2 both satisfy non-interference, with the condition that the output observation of c_1 corresponds exactly to the input observation of c_2. Unfortunately, this is

straightforward only when $\eta = id$, otherwise we have to require additivity of the output observations. This is due to the fact that, by definition, abstract non-interference checks input properties on singletons while the output of the abstract non-interference assertion for c_1 deals with properties of sets of values. In order to cope with this 'type mismatch", we need the additivity condition, as shown in the following example.

Example 4.5 *Consider the program:*

$$\begin{array}{rl} P \stackrel{def}{=} & c_1; c_2 = \\ & x := (y \bmod 2)(2x \bmod 4) + (1 - (y \bmod 2))(x \bmod 2 + 1); \\ & x := (x \bmod 2) * 4y + (1 - (x \bmod 2)) * (4y + 1) \end{array}$$

where the values are integers and the observable variable is x. Consider the property $\rho = \{\mathbb{Z}, 4\mathbb{Z}, 4\mathbb{Z}+1, 4\mathbb{Z}+2, 4\mathbb{Z}+3, \varnothing\}$ (not additive), then $[\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)c_1(\rho)]$ holds since

$$\begin{array}{l} \forall y \in 2\mathbb{Z}.\ \rho((\llbracket c_1 \rrbracket(y, \mathbb{Z}))^\circ) = \rho(\{1, 2\}) = \mathbb{Z} \text{ and} \\ \forall y \in 2\mathbb{Z}+1.\ \rho((\llbracket c_1 \rrbracket(y, \mathbb{Z}))^\circ) = \rho(\{0, 2\}) = \mathbb{Z} \end{array}$$

where we abuse notation by denoting with $\llbracket P \rrbracket$ the additive lift to sets of denotational semantics of P. On the other hand, it is simple to show that $[\rho \vdash (id)c_2(\rho)]$ since this statement leaves unchanged the abstraction of x. But if we consider the composition then we have that $[\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)P(\rho)]$ does not hold because if $y \in 2\mathbb{Z}$ then $\rho(\llbracket P \rrbracket(y, \mathbb{Z})^\circ) = \rho(\{4y, 4y+1\}) = \mathbb{Z}$ while if $y \in 2\mathbb{Z}+1$ then $\rho(\llbracket P \rrbracket(y, \mathbb{Z})^\circ) = \rho(\{4y+1\}) = 4\mathbb{Z}+1$. Note that the first statement does not satisfy abstract non-interference if we consider the disjunctive completion of ρ in output, namely its additive lift.

Finally, the next example shows that, whenever $\eta \neq id$, then requiring abstract non-interference to hold for c_2 is not sufficient to achieve soundness. For this reason, the rule requires abstract non-interference with $\eta = id$ for c_2.

Example 4.6 *Consider Par and the following program fragment P*

$$P \stackrel{def}{=} x := 4 * y^2 + 4;\ \textbf{while}\ y > 0\ \textbf{do}\ x := x \bmod 4;\ y := 0\ \textbf{endw}$$

We can prove that

$$\begin{array}{l} [\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)x := 4y^2 + 4(Par)] \text{ and} \\ [Par \vdash (id^* \times Par)\textbf{while}\ y > 0\ \textbf{do}\ x := x \bmod 4;\ y := 0\ \textbf{endw}(\rho)] \end{array}$$

where $\rho \stackrel{def}{=} \mathrm{Par} \cup \{0\}$ and x is observable. Indeed the first statement returns always an even number, while the second one, returns always even numbers if the observable input x is even, odd numbers if it is odd. On the other hand, we have that $[\mathbb{T} \vdash (id^\circ \times \mathbb{T}^*)x := 4y^2 + 4;\ c(\rho)]$, where

$$c \stackrel{def}{=} \textbf{while}\ y > 0\ \textbf{do}\ x := x \bmod 4;\ y := 0\ \textbf{endw}$$

does not hold since

$$\rho((\llbracket x := 4y^2 + 4;\ c \rrbracket(0, \mathbb{Z}))^\circ) = \rho(4) = \texttt{ev} \text{ } \textit{while}$$
$$\rho((\llbracket x := 4y^2 + 4;\ c \rrbracket(1, \mathbb{Z}))^\circ) = \rho(0) = \{0\}$$

namely abstract non-interference does not hold. At this point, note that we can prove

$$[\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)x := 4y^2 + 4(Par)] \text{ } \textit{and}$$
$$[Par \vdash (id)\textbf{while } y > 0 \textbf{ do } x := x \text{ } mod \text{ } 4;\ y := 0 \textbf{ endw}(Par)]$$

where x is the observable variable. Therefore, by applying rule **R6***, we have that $[\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)x := 4y^2 + 4;\ c(Par)]$. Indeed, for instance, if we consider $x_1 = 4$ and $x_2 = 8$ then clearly $\mathbb{T}(4) = \mathbb{T}(8) = \top$ and*

$$\begin{array}{ll} Par((\llbracket x := 4y^2 + 4;\ c \rrbracket(0, \top))^\circ) & = Par((\llbracket c \rrbracket(0, 4y^2 + 4))^\circ) \\ & = Par(4y^2 + 4) = \texttt{ev} \qquad \textit{and} \\ Par((\llbracket x := 4y^2 + 4;\ c \rrbracket(1, \top))^\circ) & = Par((\llbracket c \rrbracket(1, 4y^2 + 4))^\circ) \\ & = Par(0) = \texttt{ev} \end{array}$$

namely they are the same.

- Rule **R7** allows us to extend the results obtained with $\eta = id$ (as it happens in rule **R5**) to abstract non-interference where $\eta \neq id$. This is possible only when ϕ° distinguishes more than $(\eta)^\circ$, since intuitively this hypothesis on ϕ° allows us to apply the semantics of the program to the same set of observable inputs.
- Rule **R8** tells us that we can always make additive the output observation.
- Rule **R9** is the consequence rule, which states that we can concretize the input observation and we can abstract the output one (see Sect 3).
- The rules **R10** and **R11** say that both the least upper bound and the greatest lower bound of output observations making a program satisfy non-interference, still make the program satisfy non-interference. Rule **R12** says that the same hold for the greatest lower bound of the selection observations when these are partitioning.

We denote by $\mathcal{R}_0 = \texttt{I} \cup \{\mathbf{R1}, \ldots, \mathbf{R12}\}$ the proof system for abstract non-interference. The next result specifies that the proof system $\mathcal{R}_0$ is sound.

Lemma 4.7 *Let $\phi \in uco(\mathbb{I}_\circ)$ and $\rho \in uco(\mathbb{O}_\circ)$ additive maps such that $\forall x_1, x_2 \in \mathbb{I}.\, \phi(x_1) = \phi(x_2) \ \Rightarrow \ \rho((\llbracket P \rrbracket(x_1))^\circ) = \rho((\llbracket P \rrbracket(x_2))^\circ)$. Then $\forall X_1, X_2 \subseteq \mathbb{I}$ we have $\phi(X_1) = \phi(X_2) \ \Rightarrow \ \rho((\llbracket P \rrbracket(X_1))^\circ) = \rho((\llbracket P \rrbracket(X_2))^\circ)$*

PROOF. The following implications hold:

$$\begin{array}{lll}
\phi(X_1) = \phi(X_2) & \Leftrightarrow & \bigcup_{x_1 \in X_1} \phi(x_1) = \bigcup_{x_2 \in X_2} \phi(x_2) \text{ by additivity} \\
& \Leftrightarrow & \forall x_1 \in X_1 . \exists x_2 \in X_2 .\, \phi(x_1) = \phi(x_2) \wedge \\
& & \quad \forall x_2 \in X_2 . \exists x_1 \in X_1 .\, \phi(x_1) = \phi(x_2) \\
& \Rightarrow & \forall x_1 \in X_1 . \exists x_2 \in X_2 .\, \rho((\llbracket P \rrbracket(x_1))^\circ) = \rho((\llbracket P \rrbracket(x_2))^\circ) \wedge \\
& & \quad \forall x_2 \in X_2 . \exists x_1 \in X_1 .\, \rho((\llbracket P \rrbracket(x_1))^\circ) = \rho((\llbracket P \rrbracket(x_2))^\circ) \\
& \Leftrightarrow & \bigcup_{x_1 \in X_1} \rho((\llbracket P \rrbracket(x_1))^\circ) = \bigcup_{x_2 \in X_2} \rho((\llbracket P \rrbracket(x_2))^\circ) \\
& \Leftrightarrow & \rho((\llbracket P \rrbracket(X_1))^\circ) = \rho((\llbracket P \rrbracket(X_2))^\circ)
\end{array}$$

□

Theorem 4.8 *Let $P \in$ IMP be a program and $\phi^\circ \in uco(\mathbb{I}_\circ), \rho \in uco(\mathbb{O}_\circ)$, such that $\rho = \langle \rho_1, \ldots, \rho_n \rangle$, where $n = \left| \{\ x \in Var \mid\ x : \circ\ \} \right|$. If $\vdash_{\mathcal{R}_0} [\phi^\circ \vdash (\eta)P(\rho)]$ then $[\phi^\circ \vdash (\eta)P(\rho)]$.*

PROOF. We prove the soundness of the system inductively on the rules in Table 4. Consider $l_1, l_2 \in \mathbb{I}_\circ$ and $h_1, h_2 \in \mathbb{I}_*$. The first rule holds from Prop. 3.3 (generalization of [12][Prop. 3.7]), since we can always abstract the output observation. Let us consider **R2**. We have to prove that $\phi^\circ(l_1) = \phi^\circ(l_2)$ implies $\rho((\llbracket \mathbf{skip} \rrbracket(\eta(h_1, l_1))^\circ)) = \rho((\llbracket \mathbf{skip} \rrbracket(\eta(h_2, l_2))^\circ))$, namely implies $\rho((\eta(h_1, l_1))^\circ) = \rho((\eta(h_2, l_2))^\circ)$. By definition $\rho((\eta(h_1, l_1))^\circ) = \rho \circ (\eta)^\circ(h_1, l_1)$. Hence, the implication we require is exactly the one corresponding to the precondition $\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)$.
Consider **R3**, i.e., consider $x := e$ with $x : \circ$. The hypothesis says that $\phi^\circ(l_1) = \phi^\circ(l_2)$ implies $\rho_x(\llbracket e \rrbracket(\eta(h_1, l_1))) = \rho_x(\llbracket e \rrbracket(\eta(h_2, l_2)))$, we have to prove that $\phi^\circ(l_1) = \phi^\circ(l_2)$ implies that $\rho((\llbracket x := e \rrbracket(\eta(h_1, l_1)))^\circ) = \rho((\llbracket x := e \rrbracket(\eta(h_2, l_2)))^\circ)$. Suppose $\phi^\circ(l_1) = \phi^\circ(l_2)$ and note that, being $\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)$ then for each x and y we have that $\phi^\circ(x) = \phi^\circ(y)$ implies $\rho \circ (\eta)^\circ(x) = \rho \circ (\eta)^\circ(y)$. Let $l_1 = \langle x_1, \ldots, x, \ldots, x_n \rangle$ and $l_2 = \langle y_1, \ldots, y_n \rangle$, with $n \in \mathbb{N}$. For the condition above, the hypothesis $\phi^\circ(l_1) = \phi^\circ(l_2)$ means that $\forall i \leq n \,.\, \phi_i^\circ(x_i) = \phi_i^\circ(y_i)$ that implies that $\forall i \leq n \,.\, \rho \circ (\eta)_i^\circ(x_i) = \rho \circ (\eta)_i^\circ(y_i)$. Therefore the following equalities hold:

$$\begin{array}{lll}
\rho((\llbracket x := e \rrbracket(\eta(h_1, l_1)))^\circ) & = & \rho(\langle \eta(h_1, l_1)_1, \ldots, \llbracket e \rrbracket(\eta(h_1, l_1)), \ldots, \eta(h_1, l_1)_n \rangle) \\
& = & \langle \rho_1(\eta(h_1, l_1)_1), \ldots, \rho_x(\llbracket e \rrbracket(\eta(h_1, l_1))), \ldots, \rho_n(\eta(h_1, l_1)_n) \rangle \qquad (*) \\
& = & \langle \rho_1(\eta(h_2, l_2)_1), \ldots, \rho_x(\llbracket e \rrbracket(\eta(h_2, l_2))), \ldots, \rho_n(\eta(h_2, l_2)_n) \rangle \\
& = & \rho(\langle \eta(h_2, l_2)_1, \ldots, \llbracket e \rrbracket(\eta(h_2, l_2)), \ldots, \eta(h_2, l_2)_n \rangle) \\
& = & \rho(\llbracket x := e \rrbracket(\eta(h_2, l_2))^\circ)
\end{array}$$

where the equality $(*)$ holds since $\rho_x(\llbracket e \rrbracket(\eta(h_1, l_1))) = \rho_x(\llbracket e \rrbracket(\eta(h_2, l_2)))$ by hypothesis on e, while $\forall i .\, \rho_i(\eta(h_1, l_1)_i) = \rho_i(\eta(h_2, l_2)_i)$, being $\phi^\circ(l_1) = \phi^\circ(l_2)$ and being $\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)$ (note that the indexes from 1 to n are only the observable outputs).
Consider **R4**, suppose $\phi^\circ(l_1) = \phi^\circ(l_2)$, being $\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ (\eta)^\circ)$, we have also that $\rho(\eta(h_1, l_1)^\circ) = \rho(\eta(h_2, l_2)^\circ)$. Moreover, $\rho((\llbracket x := e \rrbracket(\eta(h_1, l_1)))^\circ)$ behaves

like **skip** on observable variables, being $x : *$, hence we have the thesis.
In order to show the soundness of **R5** we prove $c \vdash_{\mathrm{I}} \rho$, i.e., $\rho((\llbracket c \rrbracket(h,l))^\circ) = \rho(l)$, implies non-interference for the **while**, namely

$$\rho((\llbracket \textbf{while } x > 0 \textbf{ do } c \textbf{ endw} \rrbracket(h_1,l_1))^\circ) = \rho((\llbracket \textbf{while } x > 0 \textbf{ do } c \textbf{ endw} \rrbracket(h_2,l_2))^\circ)$$

for any $l_1, l_2 \in \mathbb{I}_\circ$ and $h_1, h_2 \in \mathbb{I}_*$ such that $\rho(l_1) = \rho(l_2)$. Let us denote $c_1 \stackrel{\text{def}}{=} \textbf{while } x > 0 \textbf{ do } c \textbf{ endw}$. We have to prove, by induction on the iterations of the while, that $\rho((\llbracket c_1 \rrbracket(h,l))^\circ) = \rho(l)$ for any h, l, namely we prove that this is an invariant property of the loop. If $\llbracket x > 0 \rrbracket(h,l) = \mathit{false}$, then by definition we have that $\llbracket c_1 \rrbracket = \llbracket \textbf{skip} \rrbracket$ and therefore we have the thesis by rule **R2**, being $\eta = id$ and $\phi^\circ = \rho$. Suppose now that the property holds for **while**'s with a number of loops less or equal than n, we prove it for **while**'s with $n+1$ iterations. Consider $\llbracket c_1 \rrbracket = \llbracket c; c_1 \rrbracket$ where c_1 has n iterations, we can apply the inductive hypothesis on c_1. Then

$$\begin{array}{rcl}
\rho((\llbracket c; c_1 \rrbracket(h,l))^\circ) & = & \rho((\llbracket c_1 \rrbracket(\llbracket c \rrbracket(h,l)))^\circ) \\
& = & \rho((\llbracket c_1 \rrbracket((\llbracket c \rrbracket(h,l))^*, (\llbracket c \rrbracket(h,l))^\circ))^\circ) \\
& = & \rho((\llbracket c \rrbracket(h,l))^\circ) \ \text{(by inductive hypothesis)} \\
& = & \rho(l) \ \text{(by the hypothesis of the rule on } c\text{)}
\end{array}$$

Consider rule **R6**. The hypotheses of the rule say that $\forall l_1, l_2 . \phi^\circ(l_1) = \phi^\circ(l_2)$ we have $\forall h_1, h_2 . \rho((\llbracket c_1 \rrbracket(\eta(h_1,l_1)))^\circ) = \rho((\llbracket c_1 \rrbracket(\eta(h_2,l_2)))^\circ)$ and $\forall l_1, l_2 . \rho(l_1) = \rho(l_2)$ we have $\forall h_1, h_2 \,.\, \rho_1((\llbracket c_2 \rrbracket(h_1,l_1))^\circ) = \rho_1((\llbracket c_2 \rrbracket(h_2,l_2))^\circ)$. Suppose $\phi^\circ(l_1) = \phi^\circ(l_2)$ then the following implications hold.

$$\begin{array}{rcl}
\rho_1((\llbracket c_1; c_2 \rrbracket(\eta(h_1,l_1)))^\circ) & = & \rho_1((\llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket(\eta(h_1,l_1))))^\circ) \\
& = & \rho_1((\llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket((\eta(h_1,l_1)))^*, (\llbracket c_1 \rrbracket(\eta(h_1,l_1)))^\circ))^\circ) \\
& = & \rho_1((\llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket((\eta(h_1,l_1)))^*, (\llbracket c_1 \rrbracket(\eta(h_2,l_2)))^\circ))^\circ) \\
(*) & = & \rho_1((\llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket((\eta(h_2,l_2)))^*, (\llbracket c_1 \rrbracket(\eta(h_2,l_2)))^\circ))^\circ) \\
& = & \rho_1((\llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket(\eta(h_2,l_2))))^\circ) \\
& = & \rho_1((\llbracket c_1; c_2 \rrbracket(\eta(h_2,l_2)))^\circ)
\end{array}$$

where $(*)$ holds by hypotheses and by Lemma 4.7.
In **R7** we have that $\phi^\circ(l_1) = \phi^\circ(l_2)$ implies $\rho((\llbracket c \rrbracket(h_1,l_1))^\circ) = \rho((\llbracket c \rrbracket(h_2,l_2))^\circ)$, we have also to prove that the same hypothesis implies $\rho((\llbracket c \rrbracket(\eta(h_1,l_1)))^\circ) = \rho((\llbracket c \rrbracket(\eta(h_2,l_2)))^\circ)$. The following equalities hold:

$$\begin{array}{rcl}
\rho((\llbracket c \rrbracket(\eta(h_1,l_1)))^\circ) & = & \rho\left(\left(\left(\llbracket c \rrbracket\left(\bigcup_{\langle x,y\rangle \in \eta(h_1,l_1)} (x,y)\right)\right)^\circ\right)\right) \\
& = & \rho\left(\bigcup_{\langle x,y\rangle \in \eta(h_1,l_1)} (\llbracket c \rrbracket(x,y))^\circ\right) \\
& = & \bigcup_{\langle x,y\rangle \in \eta(h_1,l_1)} \rho((\llbracket c \rrbracket(x,y))^\circ) \ \text{(Being } \rho \text{ additive)} \\
& = & \bigcup_{\langle x,y\rangle \in \eta(h_2,l_2)} \rho((\llbracket c \rrbracket(x,y))^\circ) \ (*) \\
& = & \rho((\llbracket c \rrbracket(\eta(h_2,l_2)))^\circ)
\end{array}$$

where $(*)$ holds since, by hypothesis $\phi^\circ(l_1) = \phi^\circ(l_2)$ implies $\eta(h_1, l_1)^\circ = \eta(h_2, l_2)^\circ$ and the internal part can arbitrary change by definition of non-interference. Consider **R8**. Suppose $[\phi^\circ \vdash (id)P(\rho)]$, namely $\forall h_1, h_2 \in \mathbb{I}_*$ and $\forall l_1, l_2 \in \mathbb{I}_\circ$ we have that $\phi^\circ(l_1) = \phi^\circ(l_2)$ implies $\rho((\llbracket P \rrbracket(h_1, l_1))^\circ) = \rho((\llbracket P \rrbracket(h_2, l_2))^\circ)$. At this point, since P is **deterministic**, $\llbracket P \rrbracket(h, l)^\circ$ is a singleton in $\mathbb{O}_\circ$. Therefore, from the properties of disjunctive completion, we have $\curlyvee(\rho)((\llbracket P \rrbracket(h_1, l_1))^\circ) = \rho((\llbracket P \rrbracket(h_1, l_1))^\circ) = \rho((\llbracket P \rrbracket(h_2, l_2))^\circ) = \curlyvee(\rho)((\llbracket P \rrbracket(h_1, l_1))^\circ)$, namely we have non-interference. Finally **R9**, **R10**, **R11** and **R12** hold by a straightforward generalization of [12][Prop. 3.7]. □

Next example shows a simple derivation of abstract non-interference properties, possible in our proof system.

Example 4.9 *Consider the program fragment*

$$P \stackrel{def}{=} x := 2^y;\ \textbf{while}\ y > 0\ \textbf{do}\ x := 2 * x;\ y := y - 1\ \textbf{endw}$$

where the values are naturals and x is the observable variable. First of all we note that

$$x \models [\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)2^y(\rho_1)]\ \textit{where}\ \rho_1 \stackrel{def}{=} \curlyvee\big(\{\{2\}^{\mathbb{N}}\} \cup \{\ n \mid\ n \notin \{2\}^{\mathbb{N}}\ \}\big)$$

and $\{2\}^{\mathbb{N}} \stackrel{def}{=} \{\ 2^n \mid\ n \in \mathbb{N}\ \}$, since the result is always an even number, independently from the initial value of y. This means that we can apply **R3***:*

$$\frac{x \models [\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)2^y(\rho_1)], x : \circ}{[\mathbb{T} \vdash (id^* \times \mathbb{T}^\circ)x := 2^y(\rho_1)]}$$

Consider the **while***-statement, denoted by c, and consider the closure operator $\rho_2 \stackrel{def}{=} \curlyvee(\{\ n\{2\}^{\mathbb{N}} \mid\ n \in \mathbb{N}\ odd\ \})$. We note that $\langle 2 * x, x\rangle \vdash_{\text{I}} \rho_2$, since the operation $2 * x$ does not change the property $n\{2\}^{\mathbb{N}}$ of the initial value of x, namely it does not change the odd factor of x. Therefore, we can apply* **I4** *to the observable assignment and* **I3** *for the internal assignment:*

$$\frac{\langle 2 * x, x\rangle \vdash_{\text{I}} \rho_2, x : \circ}{x := 2 * x \vdash_{\text{I}} \rho_2} \qquad \frac{h : *}{y := y - 1 \vdash_{\text{I}} \rho_2}$$

and therefore by applying **I5** *we obtain*

$$\frac{x := 2 * x \vdash_{\text{I}} \rho_2, y := y - 1 \vdash_{\text{I}} \rho_2}{x := 2 * x;\ y := y - 1 \vdash_{\text{I}} \rho_2}$$

Now we can apply **R5**

$$\frac{x := 2 * x;\ y := y - 1 \vdash_{\text{I}} \rho_2}{[\rho_2 \vdash (id)\textbf{while}\ y > 0\ \textbf{do}\ x := 2 * x;\ y := y - 1\ \textbf{endw}(\rho_2)]}$$

and therefore we use **R6***:*

$$\frac{[\mathbb{T}^\circ \vdash (id^* \times \mathbb{T}^\circ)x := 2^y(\rho_1)], [\rho_2 \vdash (id)c(\rho_2)]}{(\mathbb{T})P(\rho_2)}$$

4.3 Complete proof system

Unfortunately, the system $\mathcal{R}_0$ is not complete, and in particular **R6** is the rule that introduces incompleteness.

Example 4.10 *Consider the property Par, and the program P in the Example 4.6, where the values are integer and the observable variable is x. Let us denote the* **while** *statement as $c \stackrel{def}{=}$* **while** $y > 0$ **do** $x := x \bmod 4$; $y := 0$ **endw**. *We can prove that*

$$[\mathbb{T} \vdash (id)x := 4y^2 + 4(\rho_1)] \text{ and } [\mathbb{T} \vdash (id)P(\rho_1)] \text{ hold}$$

where ρ_1 is the closure which is not able to distinguish even numbers, i.e., $\rho_1 = \curlyvee(\{\mathtt{ev}\} \cup \{\ \{n\} \mid n \text{ odd}\ \})$. These facts hold since the result of the assignment is always an even number multiple of 4, independently from the value of y (so the first fact holds). At this point, the **while** *receives a multiple of 4 and therefore the result is always 0, implying the second fact. On the other hand, we have that it does not hold*

$$[\rho_1 \vdash (id)c(\rho_1)]$$

since without the assignment, the **while** *can receive any number, in particular it can receive, as inputs, numbers that are not multiples of 4. For these numbers the statement does not satisfy abstract non-interference, for instance $\rho_1([\![c]\!](0,5)^\circ) = 5 \neq \rho_1([\![c]\!](1,5)^\circ) = 1$. This means that $\not\vdash_{\mathcal{R}_0} [\mathbb{T} \vdash (id)P(\rho_1)]$*[2]*.*

In abstract non-interference, the systematic construction of secret kernels (see Sect.3), plays a key role for making the proof systems $\mathcal{R}_0$ complete. Completeness is here achieved by respectively including the following *semantic rule*:

$$\textbf{R0:}\ \frac{[\phi^\circ \vdash (\eta)[\![c]\!](id)] \sqsubseteq \rho}{[\phi^\circ \vdash (\eta)c(\rho)]}$$

The semantic rule **R0** derives from [12][Th. 5.5]. It states that, given a program c and the input observations ϕ° and η we can derive the most concrete output observation that makes the program satisfy abstract non-interference. This corresponds precisely to finding the strongest post-condition (viz., the most concrete abstract domain) for the program c with precondition ϕ° and η, such that abstract non-interference holds. This is a "semantic rule", because it involves the construction of the abstract domain $[\phi^\circ \vdash (\eta)[\![c]\!](id)]$, which is equivalent to

[2]This example does not imply that there not exist a ρ_2 such that abstract non-interference holds, but it shows that we cannot derive, in this case, an abstract non-interference property that holds by using this system.

compute the concrete semantics of the command c. However, this rule allows us to include in the abstract non-interference proof, assertions which can be systematically derived as an abstract domain transformation. The idea is to use this rule for deriving some starting properties, for example for expressions, or for some simple statements, and then to use the proof system for deriving the non-interference property for the whole program. Let $\mathcal{R} = \mathcal{R}_0 \cup \{\mathbf{R0}\}$. It is clear that rule **R0** makes the proof systems $\mathcal{R}$ complete.

Corollary 4.11 *The proof systems $\mathcal{R}$ is complete.*

Proof. If $[\phi^\circ \vdash (\eta)P(\rho)]$ then $\rho \sqsupseteq [\phi^\circ \vdash (\eta)[\![P]\!](id)]$ by [12][Th. 5.5]. Therefore for by Rule **R0** we have that $\vdash_{\mathcal{R}} [\phi^\circ \vdash (\eta)P(\rho)]$. □

4.4 ANI for non-deterministic systems

In the following, we consider the simple imperative language with non deterministic choice, Nd-Imp, introduced in Sec. 2.3. As usual $[\![P]\!]$ denotes the input/output relation for the program P also in the non-deterministic case, therefore $[\![P]\!](s)$ denotes the set of all states reachable by executing P starting from the state s.

In this context, consider the notion of *possibilistic* non-interference [26] for non-deterministic programs: A program is secure if given two states s_1 and s_2 such that $s_1^{\mathtt{L}} = s_2^{\mathtt{L}}$, then for each computation σ with $\sigma_{\vdash}^{\mathtt{L}} = s_1^{\mathtt{L}}$ there exists a computation δ with $\delta_{\vdash}^{\mathtt{L}} = s_2^{\mathtt{L}}$, such that $\sigma_{\dashv}^{\mathtt{L}} = \delta_{\dashv}^{\mathtt{L}}$ (see Sec. 2.3). This notion can be formulated as in Def. 3.1 with only semantic difference that now $[\![P]\!](s)^{\mathtt{L}}$ is a set of values instead of a single value. Anyway, the generalization is not so straightforward. Indeed if we don't consider additive closures for the output observation, the notion of non-interference as given above, is not precise. In fact, missing additivity means that the property of a set is not the union of the properties of its elements. In the context of non-interference, this means that the collection of all observations of the single computations, does not correspond to the observation of the set of all possible results. Indeed, we recall that possibilistic non-interference is based on the assumption that the attacker can observe and collect all possible system behaviours. Therefore, if it is able to observe the property ρ of the output, then the natural non-deterministic extension of abstract non-interference would say that the attacker can collect the set of all ρ observations of the possible system behaviours, which is in general different from the ρ property of the set of all possible system behaviours. Therefore, in order to define abstract non-interference for non-deterministic systems simply by considering the non-deterministic denotational semantics as defined by Cousot [7], we have to consider only additive properties for the output observation. Therefore, when ρ is *additive*, i.e., $\rho = \bigvee(\rho)$, we define abstract non-interference exactly as we have done for deterministic systems, as follows:

A program P is *secure* if
$$\forall x_1, x_2 \in \mathbb{I}\,.\ \phi(x_1) = \phi(x_2) \ \Rightarrow\ \rho(([\![P]\!](\eta(x_1)))^\circ) = \rho(([\![P]\!](\eta(x_2)))^\circ)$$

Exactly as it happens in the deterministic case, we have to require some restrictions on ϕ in order to guarantee the sequential compositionality of the abstract non-interference properties. Hence, also in this case, ϕ can only select observable data, and therefore it has the form $\phi = \mathbb{T}^* \times \phi^\circ$, and we choose to explicitly denote only the observable part, using exactly the same notation as in the deterministic case.

We now extend the proof system for deterministic programs in order to cope with non-deterministic ones. We first derive the rule for the non-deterministic choice in the proof system $\mathtt{I}$:

$$\textbf{I8:}\ \frac{\forall i \in I\ .\ c_i \vdash_{\mathtt{I}} \rho_i}{\Box_i c_i \vdash_{\mathtt{I}} \bigsqcup_{i\in I} \rho_i}$$

Rule **I8** controls the non-deterministic choice in a rather standard way. Indeed, it says that an invariant property for a non-deterministic choice is the most abstract invariant among the ones for all programs involved in the non-deterministic choice. At this point we have to modify the proof system $\mathcal{R}$. The problem here is that it is not sufficient to add the rule for non-deterministic choice, since the fact that the denotational semantics returns a set of values instead of a singleton induces some new considerations on rule **R6**.

Example 4.12 *Consider the program:*

$$P = c_1;\ c_2 = \begin{array}{l} l := 1 - (h\ mod\ 2)\ \Box\ l := 2 * (h\ mod\ 2) + 2 * (1 - (h\ mod\ 2)); \\ l := (l\ mod\ 2) * 4h + (1 - (l\ mod\ 2)) * (4h + 1) \end{array}$$

with typing $h : *$ and $l : \circ$. Consider the property observing the modulus in the division by 4: $\rho = \{\mathbb{Z}, 4\mathbb{Z}, 4\mathbb{Z}+1, 4\mathbb{Z}+2, 4\mathbb{Z}+3, \varnothing\}$ (not additive), then we can show that $[\mathbb{T} \vdash (id)c_1(\rho)]$ since

$$\begin{array}{l} \forall h \in \mathtt{ev}.\ \forall l \in \mathbb{Z}\ \rho((\llbracket c_1 \rrbracket(h,l))^\circ) = \rho(\{1,2\}) = \mathbb{Z} \text{ and} \\ \forall h \in \mathtt{od}.\ \forall l \in \mathbb{Z}\ \rho((\llbracket c_1 \rrbracket(h,l))^\circ) = \rho(\{0,2\}) = \mathbb{Z} \end{array}$$

On the other hand, it is simple to show that $[\rho \vdash (id)c_2(\rho)]$ since the abstraction of l does not depend on h. But if we consider the composition then we have that $[\mathbb{T} \vdash (id)P(\rho)]$ does not hold because

$$\begin{array}{l} \forall h \in \mathtt{ev}.\ \forall l \in \mathbb{Z}\ \rho((\llbracket P \rrbracket(h,l))^\circ) = \rho(\{4h, 4h+1\}) = \mathbb{Z} \\ \forall h \in \mathtt{od}.\ \forall l \in \mathbb{Z}\ \rho((\llbracket P \rrbracket(h,l))^\circ) = \rho(\{4h+1\}) = 4\mathbb{Z}+1 \end{array}$$

while rule **R6** *would infer that $[\mathbb{T} \vdash (id)P(\rho)]$ holds, and the problem lies on the fact that ρ is not additive.*

Therefore we replace rule **R6** with **R′6**

$$\textbf{R′6:}\ \frac{[\phi^\circ \vdash (\eta)c_1(\rho)],\ [\rho \vdash (id)c_2(\rho_1)],\ \rho, \rho_1 \text{ additive}}{[\phi^\circ \vdash (\eta)c_1; c_2(\rho_1)]}$$

We can observe that rule **R8** becomes useless since, in the non-deterministic context, we need output additive observations, and this means that we have no distinction between ρ and $\curlyvee(\rho)$. Finally, we introduce the rule **R13** for the non-deterministic choice.

$$\textbf{R13:}\ \frac{\forall i \in I\ .\ [\phi_i^\circ \vdash (\eta)c_i(\rho_i)]}{[\bigsqcap_{i\in I}\phi_i^\circ \vdash (\eta)\Box_i c_i(\bigsqcup_{i\in I}\rho_i)]}$$

R13 says that, if we have a non-deterministic choice among the elements of a set of programs, then this non-deterministic choice satisfies non-interference for the observer characterized, in input, by the greatest lower bound of input observations for which the elements of the set satisfy non-interference, and in output by the least upper bound of output observations of the same elements. For instance, note that if we have $c \stackrel{\text{def}}{=} l := 2 * h \,\Box\, l := 2h + l$, where it is worth noting that, if $\rho = \curlyvee\big(\{\mathtt{ev}\} \cup \big\{\ \{n\}\ \big|\ n \text{ odd}\ \big\}\big)$, we obtain $[\rho \vdash (id)l := 2 * h(\rho)]$ and $[Par \vdash (id)l := 2h + l(Par)]$. Clearly the execution of c has to guarantee non interference independently from the statement that is executed, so we have $[\rho \vdash (id)c(Par)]$. For all other rules we have simply to add the requirement that the output observations are additive when the program is non-deterministic.

Lemma 4.13 *Let $\eta \in uco(\wp(\mathbb{I}_\circ))$ and $\rho \in uco(\wp(\mathbb{O}_\circ))$ additive, and suppose $[\eta \vdash (id)P(\rho)]$, then we have that for each $L_1, L_2 \in \wp(\mathbb{I}_\circ)$ and $H_1, H_2 \in \wp(\mathbb{I}_*)$ if $\eta(L_1) = \eta(L_2)$ then $\rho((\llbracket P \rrbracket(H_1, L_1))^\circ) = \rho((\llbracket P \rrbracket(H_2, L_2))^\circ)$*

Proof. Being ρ additive we have

$$\rho((\llbracket P \rrbracket(H_1, L_1))^\circ) = \bigcup_{h_1\in H_1, l_1 \in L_1} \rho((\llbracket P \rrbracket(h_1, l_1))^\circ).$$

Since also η is additive, we have that $\eta(L_1) = \eta(L_2)$ implies that for each $l_1 \in Ł_1$ there exists $l_2 \in L_2$ such that $\eta(l_1) = \eta(l_2)$. Namely we have that

$$\bigcup_{h_1\in H_1, l_1 \in L_1} \rho((\llbracket P \rrbracket(h_1, l_1))^\circ) \subseteq \bigcup_{h_2\in H_2, l_2 \in L_2} \rho((\llbracket P \rrbracket(h_2, l_2))^\circ)$$

since $[\eta \vdash (id)P(\rho)]$. Viceversa we can prove the other inclusion in a similar way, therefore we have that

$$\rho((\llbracket P \rrbracket(H_1, L_1))^\circ) = \rho((\llbracket P \rrbracket(H_2, L_2))^\circ).$$

□

Theorem 4.14 *The proof system $\mathcal{R}_0^{\mathrm{ND}} \stackrel{def}{=} \mathcal{R}_0 \smallsetminus \{\mathbf{R6}\} \cup \{\mathbf{I8}, \mathbf{R'6}, \mathbf{R13}\}$ (where the additivity condition on the output observation is added to all the rules) is sound and the proof system $\mathcal{R}^{\mathrm{ND}} \stackrel{def}{=} \mathcal{R}_0^{\mathrm{ND}} \cup \{\mathbf{R0}\}$ is complete.*

PROOF. The completeness is straightforward by the presence of the rule **R0** (see Corollary 4.11).
Correctness of **I8** is straightforward from rule **I7**. In order to prove correctness of **R′6** we have to show that whenever the premises of the rule hold then the consequence holds as well. Consider ρ and ρ_1 additive, namely $\rho = \curlyvee(\rho)$ and $\rho_1 = \curlyvee(\rho_1)$. Then we suppose that $[\phi^\circ \vdash (\eta)c_1(\rho)]$ and that $[\rho \vdash (id)c_2(\rho_1)]$, namely if $\phi^\circ(l_1) = \phi^\circ(l_2)$ then $\rho((\llbracket c_1 \rrbracket(\eta(h_1, l_1)))^\circ) = \rho((\llbracket c_1 \rrbracket(\eta(l_2, h_2)))^\circ)$ and if $\rho(l_1) = \rho(l_2)$ then $\rho_1((\llbracket c_2 \rrbracket(h_1, l_1))^\circ) = \rho_1((\llbracket c_2 \rrbracket(l_2, h_2))^\circ)$. We have to prove that if $\phi^\circ(l_1) = \phi^\circ(l_2)$ then we have $\rho_1((\llbracket c_1; c_2 \rrbracket(\eta(h_1, l_1)))^\circ) = \rho_1((\llbracket c_1; c_2 \rrbracket(\eta(l_2, h_2)))^\circ)$. Hence suppose $\phi^\circ(l_1) = \phi^\circ(l_2)$, the following equalities hold:

$$\begin{aligned}
\rho_1((\llbracket c_1; c_2 \rrbracket(\eta(h_1, l_1)))^\circ) &= \rho_1((\llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket(\eta(h_1, l_1))))^\circ) \\
&= \rho((\llbracket c_2 \rrbracket((\llbracket c_1 \rrbracket(\eta(h_1, l_1)))^*, (\llbracket c_1 \rrbracket(\eta(h_1, l_1)))^\circ))^\circ) \\
&= \rho((\llbracket c_2 \rrbracket((\llbracket c_1 \rrbracket(\eta(h_2, l_2)))^*, (\llbracket c_1 \rrbracket(\eta(h_2, l_2)))^\circ))^\circ) \qquad \text{(By Lemma 4.13)} \\
&= \rho((\llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket(\eta(h_2, l_2))))^\circ) = \rho((\llbracket c_1; c_2 \rrbracket(\eta(h_2, l_2)))^\circ)
\end{aligned}$$

and so we have non-interference.
R13 is sound since $\llbracket \Box_i c_{i \in I} \rrbracket = \llbracket c_k \rrbracket$ for some $k \in I$, and $[\phi_k^\circ \vdash (\eta)c_k(\rho_k)]$, that holds by hypothesis, implies by **R9**, that $[\sqcap_i \phi_i^\circ \vdash (\eta)c_k(\sqcup_i \rho_i)]$.
□

5 An application to language-based security

In this section, we focus on the instantiation, of the proof system, introduced in the previous section, to the context of language-based security. In particular, the idea is to derive a proof system for each abstract non-interference notion introduced in [12]: Narrow and Abstract Non-Interference. The fact that we are considering security adds a new constraint on the considered abstractions: in this case also η is an attribute independent abstraction, namely it is composed by an internal (here private, denoted `H`) and observable (here public, denoted `L`) part. In other words, it cannot describe relations between internal and observable data. Moreover, in sake of simplicity, we only consider deterministic programs. In this way, we have only two cases. The first one is narrow (abstract) non-interference, which consists in considering an observer (here called attacker) that can only observe the I/O behaviour of programs by means of an abstraction ϕ° in input and an abstraction ρ in output. In this case, the property η is the identity, since the semantics of programs is abstracted only in the output. The other case is called abstract non-interference (for security) and it considers $\eta^\circ = \phi^\circ$ and $\eta^* = id^*$. Namely we characterise again the attacker with only two abstractions, an input and an output one, and for this reason the input observational capability of the attacker is the same in the selection of the inputs and in the observation of data.

Note that, the restrictions we introduced in the general framework on ϕ imply the impossibility to consider declassification (via allowing) [20, 3] in our

proof system. Declassified abstract non-interference (via allowing) considers an abstraction of private input characterising what private property we allow to flow in the observable part. This abstraction, in our context corresponds to ϕ^*. In this case, for example narrow abstract non-interference becomes:

$$\forall h_1, h_2 : \mathtt{H}, l_1, l_2 : \mathtt{L}. \phi^\circ(l_1) = \phi^\circ(l_2) \ \wedge \ \phi^*(h_1) = \phi^*(h_2) \\ \Rightarrow \ \rho((\llbracket P \rrbracket(h_1, l_1))^\circ) = \rho((\llbracket P \rrbracket(h_2, l_2))^\circ)$$

At this point, it is possible to verify that this notion is not compositional wrt sequential composition, exactly as we noticed in the general case. This observation, therefore, excludes the possibility of generating a similar proof system for this kind of declassified abstract non-interference.
The things are different for the other kind of declassification introduced in the context of abstract non-interference [12, 20]: Declassification via blocking. In this case we have again an abstraction on the private input, but it represents what we don't want to flow in the observable data. In our context this corresponds to the property η^*. Namely, abstract non-interference declassified via blocking is precisely the property $[\phi^\circ \vdash (\eta^* \times \phi^\circ)P(\rho)]$, which perfectly fits in the general proof system introduced in the previous section, simply with some restrictions on η.

5.1 Proof system for Narrow (Abstract) Non-Interference

Consider first the situation where the observer can only observe, and therefore abstract, the I/O behaviour of the program. In particular, we have the input observation of the input, which is ϕ°, and the output observation ρ, while we do not have an abstraction of the semantics in input, namely $\eta = id$. In sake of simplicity, and for coherence with previous works [12] we call this particular instantiation narrow, and we denote it $[\phi^\circ]P(\rho)$. In the following we explain the meaning of the different rules in this particular context, and we show how certain rules change due to the new constraints. Moreover, being in the context of security, the internal data will be called *private* and denoted as $\mathtt{H}$, while the observable data will be called *public* and denoted $\mathtt{L}$. Rules from R_N1 to R_N4 and from R_N8 to R_N12 are trivial instantiations of the corresponding rules in $\mathcal{R}$, with the only observation that, being $\eta = id$, $\rho \circ \eta^\circ$ becomes ρ. In R_N6 we lose the additivity condition, since here we consider only $\eta = id$. R_N5 is exactly **R5**, while we have not the instantiation of **R7** since this rule becomes meaningless when $\eta = id$.

We denote by $\mathcal{R}_0^N = \mathtt{I} \cup \{R_N1, \ldots, R_N11\}$ the proof system for (narrow) abstract non-interference.

Example 5.1 *Consider the closure* Par*, and the program:*

$$P \stackrel{def}{=} l := 2 * h; \ \textbf{while } h > 0 \textbf{ do } l := l + 2; \ h := h - 1 \textbf{ endw}$$

with security typing $h : \mathtt{H}$ *and* $l : \mathtt{L}$*, and with* $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$*. We can show that* $[\mathbb{T}]2 * h(\rho_1)$ *where* ρ_1 *is the closure which is not able to distinguish even*

$$R_N1\textbf{:}\ [\phi^\circ]c(\mathbb{T}) \qquad R_N2\textbf{:}\ \frac{\Pi(\phi^\circ) \sqsubseteq \Pi(\rho)}{[\phi^\circ]\textbf{skip}(\rho)} \qquad R_N3\textbf{:}\ \frac{x \models [\phi^\circ]e(\rho),\ [\Pi(\phi^\circ) \sqsubseteq \Pi(\rho)],\ x : \mathtt{L}}{[\phi^\circ]x := e(\rho)}$$

$$R_N4\textbf{:}\ \frac{x : \mathtt{H},\ \Pi(\phi^\circ) \sqsubseteq \Pi(\rho)}{[\phi^\circ]x := e(\rho)} \qquad R_N5\textbf{:}\ \frac{c \vdash_{\mathtt{I}} \rho}{[\rho]\textbf{while}\ x > 0\ \textbf{do}\ c\ \textbf{endw}(\rho)} \qquad R_N6\textbf{:}\ \frac{[\phi^\circ]c_1(\rho),\ [\rho]c_2(\rho_1)}{[\phi^\circ]c_1; c_2(\rho_1)}$$

$$R_N8\textbf{:}\ \frac{[\phi^\circ]c(\rho)}{[\phi^\circ]c(\curlyvee(\rho))} \qquad R_N9\textbf{:}\ \frac{[\phi_1^\circ]c(\rho_1),\ \phi^\circ \sqsubseteq \phi_1^\circ,\ \rho_1 \sqsubseteq \rho}{[\phi^\circ]c(\rho)}$$

$$R_N10\textbf{:}\ \frac{\forall i \in I\,.\,[\phi_1^\circ]c(\rho),\ \phi_i^\circ \text{ partitioning}}{[\bigsqcup_{i\in I}\phi_i^\circ]c(\rho)} \qquad R_N11\textbf{:}\ \frac{\forall i \in I\,.\,[\phi^\circ]c(\rho_i)}{[\phi^\circ]c(\bigsqcup_{i\in I}\rho_i)} \qquad R_N12\textbf{:}\ \frac{\forall i \in I\,.\,[\phi^\circ]c(\rho_i)}{[\phi^\circ]c(\bigsqcap_{i\in I}\rho_i)}$$

Table 4: Axiomatic narrow (abstract) non-interference

numbers, i.e., $\rho_1 = \curlyvee\big(\{\mathtt{ev}\} \cup \big\{\ \{n\}\ \big|\ n \text{ odd}\ \big\}\big)$*. Therefore, by* R_N3*, we obtain* $[\mathbb{T}]l := 2 * h(\rho_1)$ *(note that, since there is only one low variable we ignore the condition* $\Pi(\eta) \sqsubseteq \Pi(\rho)$*). Consider now the* **while***-statement. We note that the operation* $l + 2$ *leaves unchanged the parity of* l*, this means that if the input is even the the output is even, and similarly if it is odd. Namely for each* n *such that* $Par(n) = Par(l)$ *then* $Par([\![l + 2]\!](h, n)) = Par(n + 2) = Par(n) = Par(l)$*. Therefore* $\langle l + 2, l\rangle \vdash_{\mathtt{I}} Par$ *which implies*

$$\frac{\langle l + 2, l\rangle \vdash_{\mathtt{I}} Par}{l := l + 2 \vdash_{\mathtt{I}} Par} \qquad \frac{h : \mathtt{H}}{h := h + 1 \vdash_{\mathtt{I}} Par}$$

Therefore, by **I5***, we have that* $l := l + 2;\ h := h - 1 \vdash_{\mathtt{I}} Par$*. Now we can apply rule* R_N6 *obtaining*

$$\frac{l := l + 2;\ h := h - 1 \vdash_{\mathtt{I}} Par}{[Par]\textbf{while}\ h > 0\ \textbf{do}\ l := l + 2;\ h := h - 1\ \textbf{endw}(Par)}$$

Finally, note that $\rho_1 \sqsubseteq Par$ *hence by* R_N8 *we have also that* $[\mathbb{T}]l := 2 * h(Par)$*, therefore we can apply rule* R_N6 *and we obtain that* $[\mathbb{T}]P(Par)$*.*

5.2 Proof system for Abstract Non-Interference

Consider now the situation where the observer can also analyse the code, and therefore it can abstract the semantics of the program. In particular, we have the input observation ϕ°, which corresponds to the input analysis of the semantics, i.e., $\eta^\circ = \phi^\circ$, and the output observation ρ, while we do not have an abstraction of the private input semantics, namely $\eta^* = id^*$. In sake of simplicity, and for coherence with previous works [12] we call this particular instantiation generically abstract, and we denote it $(\phi^\circ)P(\rho)$. In the following we explain the meaning of the different rules in this particular context, and we show how certain rules change due to the new constraints.

$$R_A1\textbf{:}\ (\phi^\circ)c(\mathbb{T}) \qquad R_A2\textbf{:}\ (\phi^\circ)\mathbf{skip}(\rho) \qquad R_A3\textbf{:}\ \frac{x \models (\phi^\circ)e(\rho),\ x : \mathtt{L}}{(\phi^\circ)x := e(\rho)} \qquad R_A4\textbf{:}\ \frac{x : \mathtt{H}}{(\phi^\circ)x := e(\rho)}$$

$$R_A5\textbf{:}\ \frac{c \vdash_{\mathtt{I}} \rho,\ x : \mathtt{H}}{(\rho)\mathbf{while}\ x > 0\ \mathbf{do}\ c\ \mathbf{endw}(\rho)} \qquad R_A5\mathbf{bis}\textbf{:}\ \frac{[\rho]c(\rho),\ \rho \text{ additive},\ x : \mathtt{L}}{[\rho]\mathbf{while}\ x > 0\ \mathbf{do}\ c\ \mathbf{endw}(\rho)}$$

$$R_A6\textbf{:}\ \frac{(\phi^\circ)c_1(\rho),\ [\rho]c_2(\rho_1),\ \rho, \rho_1 \text{ additive}}{(\phi^\circ)c_1; c_2(\rho)} \qquad R_A9\textbf{:}\ \frac{(\phi^\circ)c(\rho_1),\ \rho_1 \sqsubseteq \rho}{(\phi^\circ)c(\rho)}$$

$$R_A11\textbf{:}\ \frac{\forall i \in I\,.\,(\phi^\circ)c(\rho_i)}{(\phi^\circ)c(\bigsqcup_{i\in I}\rho_i)} \qquad R_A12\textbf{:}\ \frac{\forall i \in I\,.\,(\phi^\circ)c(\rho_i)}{(\phi^\circ)c(\bigsqcap_{i\in I}\rho_i)} \qquad R_A13\textbf{:}\ \frac{[\phi^\circ]c(\rho)}{(\phi^\circ)c(\rho)}$$

Table 5: Axiomatic abstract non-interference

Rules R_A1, R_A9, R_A11 e R_A12 are trivial instantiations of the corresponding rules in $\mathcal{R}$. Also the rules from R_A3 to R_A6 are trivial instantiations of the corresponding rules, noting that $\rho \circ \eta^\circ = \rho \circ \phi^\circ$, and therefore $\Pi(\phi^\circ) \sqsubseteq \Pi(\rho \circ \phi^\circ)$ trivially holds since $\phi^\circ \sqsubseteq \rho \circ \phi^\circ$. As far as the **while** is concerned the new constraints make the analysis more precise. In particular rule **R5** can be instantiated only when the guard is private, namely when implicit flows are possible. Instead, when the guard is public, we can consider a new weaker rule $R_A5\mathbf{bis}$. This rule was not possible before, because the possible difference between ϕ° and η° could cause *deceptive* interference [12]. Finally, rule R_A13 describe the relation between narrow and abstract non-interference [12]. The following example shows the difference between narrow and abstract non-interference properties for loops.

Example 5.2 *Consider the program fragment:*

$$P \stackrel{def}{=} \mathbf{while}\ l < 2\ \mathbf{do}\ l := 2l\ \mathbf{endw}$$

and consider the non-interference property $[Sign]P(Par)$*. Then we can note that, this property does not hold even if it holds for the body of the while. Indeed, we can trivially verify that* $[Sign]l := 2l(Par)$*, while if we consider* $l_1 = 1$ *and* $l_2 = 3$*, then* $Sign(1) = Sign(3)$*, while* $Par((\llbracket P \rrbracket(h_1, 1))^{\mathtt{L}}) = Par(2) = \mathtt{ev} \neq \mathtt{od} = Par(3) = Par((\llbracket P \rrbracket(h_2, 3))^{\mathtt{L}})$*. This means that, in the general case, even if the guard is observable we need the condition about invariant properties.*
Consider now $(Sign)P(Par)$*:* $Par((\llbracket P \rrbracket(h_1, Sign(1)))^{\mathtt{L}}) = Par((\llbracket P \rrbracket(h_1, +))^{\mathtt{L}}) = Par((\llbracket P \rrbracket(h_2, +))^{\mathtt{L}}) = Par((\llbracket P \rrbracket(h_2, Sign(3)))^{\mathtt{L}})$*. Hence, in the abstract case we can be more precise removing the invariant condition when the guard is observable.*

R_A8 is meaningless, since it concerns only narrow non-interference. Finally **R9** is not applicable here, where $\phi = \eta$, because it requires the independence between ϕ and η, since in the rule ϕ can change while η remain fixed.

The proof system for abstract non-interference in Table 5 is denoted $\mathcal{R}_0^A = \mathcal{R}_0^N \cup \{R_A1, \ldots, R_A12\}$.

Theorem 5.3 *Let $P \in \text{IMP}$ be a program and $\eta, \rho \in uco(\mathbb{V}^{\mathtt{L}})$, such that $\rho = \langle \rho_1, \ldots, \rho_n \rangle$, where $n = \left|\left\{ x \in Var \middle| x : \mathtt{L} \right\}\right|$. If $\vdash_{\mathcal{A}_0} (\eta)P(\rho)$ then $(\eta)P(\rho)$.*

PROOF. Clearly, all rules but R_A5**bis** and R_A13 are trivial instantiations of the corresponding rules in the system $\mathcal{R}$, hence we inherit their soundness. R_A13 is a trivial consequence of the fact that $[\eta]P(\rho)$ implies $(\eta)P(\rho)$ [12]. Hence, we have only to prove soundness of R_A5**bis**. Consider $[\rho]c(\rho)$, and $\rho(l_1) = \rho(l_2)$, then we have to prove that $\rho((\llbracket \textbf{while } x > 0 \textbf{ do } c \textbf{ endw} \rrbracket(h_1, l_1))^{\mathtt{L}}) = \rho((\llbracket \textbf{while } x > 0 \textbf{ do } c \textbf{ endw} \rrbracket(h_2, l_2))^{\mathtt{L}})$. The only difference with rule **R5** is that, since the guard is public, the number of iterations is the same in both the cases, hence we have only to show that non-interference is preserved by composition. This can be simply showed by induction on the number of iterations of the while and by using Rule R_N6. □

The following example shows that also the proof system $\mathcal{R}_0^A$ for abstract non-interference in Table 5 is not complete.

Example 5.4 *Consider the closure $\rho \stackrel{def}{=} \{\mathbb{Z}, 2\mathbb{Z}, 4\mathbb{Z}, \varnothing\}$ and consider the program*

$$P \stackrel{def}{=} \textbf{while } h > 0 \textbf{ do } l := (l \ mod \ 4) * (l \div 4);\ h := 0 \textbf{ endw}$$

with security typing $h : \mathtt{H}$ and $l : \mathtt{L}$, and with $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$. Note that $(\rho)P(\rho)$ since, for example, $\rho((\llbracket P \rrbracket(1, 2\mathbb{Z}))^{\mathtt{L}}) = 2\mathbb{Z} = \rho((\llbracket P \rrbracket(0, 2\mathbb{Z}))^{\mathtt{L}})$. But we have that $P \vdash_{\scriptscriptstyle 1} \rho$ does not hold since $\rho((\llbracket P \rrbracket(1, 2))^{\mathtt{L}}) = \rho(0) = 4\mathbb{Z} \neq \rho(2) = 2\mathbb{Z}$.

The example above shows that **A5** introduces incompleteness in the system, but it is not the only such a rule. In particular, by the same argument used in Example 4.10 for **R6**, R_A6 introduces also incompleteness. Even R_A5**bis** introduces incompleteness, since the guard of the while can avoid interferences that may happen in the body, as shown in the following example.

Example 5.5 *Consider $\rho \stackrel{def}{=} \{\mathbb{Z}, \{0\}, \mathtt{ev}_0, \mathtt{od}, \varnothing\}$, where $\mathtt{ev}_0 \stackrel{def}{=} \mathtt{ev} \smallsetminus \{0\}$, and*

$$P \stackrel{def}{=} \textbf{while } l_1 > 0 \textbf{ do } l_2 := iszero(l_1) * h^2;\ l_1 := 0 \textbf{ endw}$$

with security typing $h : \mathtt{H}$ and $l_1, l_2 : \mathtt{L}$, and with

$$iszero(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

Then we can prove that it does not hold that

$$[\rho]l_2 := iszero(l_1) * h^2;\ l_1 := 0(\rho)$$

since, if we take the low input $\langle 0, 2\rangle, \langle 0, 8\rangle \in \langle 0, \mathtt{ev}_0\rangle$ then we have

$$\rho((\llbracket l_2 := iszero(l_1) * h^2;\ l_1 := 0 \rrbracket(1, \langle 0, 2\rangle))^{\mathtt{L}}) = \rho(\langle 0, 1\rangle) = \langle 0, \mathtt{od}\rangle \neq$$
$$\rho((\llbracket l_2 := iszero(l_1) * h^2;\ l_1 := 0 \rrbracket(2, \langle 0, 8\rangle))^{\mathtt{L}}) = \rho(\langle 0, 4\rangle) = \langle 0, \mathtt{ev}_0\rangle$$

But it is worth noting that $[\rho]P(\rho)$ since, for example, $\rho((\llbracket P \rrbracket(h, \langle 0, 2\rangle))^{\mathtt{L}}) = \langle 0, \mathtt{ev}_0\rangle$ and $\rho((\llbracket P \rrbracket(h, \langle 4, 8\rangle))^{\mathtt{L}}) = \langle 0, 0\rangle$.

The next example shows that $\mathcal{R}_0^A$ is strictly weaker than $\mathcal{R}_0^N$. We show that if $[\eta]P(\rho)$ and $\vdash_{\mathcal{R}_0^A} (\eta)P(\rho)$, the fact that $[\eta]P(\rho) \Rightarrow (\eta)P(\rho)$ does not imply that $\vdash_{\mathcal{R}_0^N} [\eta]P(\rho)$.

Example 5.6 *Consider the property* Par *and the program:*

$$P \stackrel{def}{=} h := h + 1; l := 2 * h$$

with security typing $h : \mathtt{H}$ *and* $l : \mathtt{L}$*, and with* $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$*. Note that* $[Sign]P(Par)$ *since*

$$\forall l \in \mathbb{V}^{\mathtt{L}},\ h \in \mathbb{V}^{\mathtt{H}} \text{ we have } Par((\llbracket P \rrbracket(h,l))^{\mathtt{L}}) = Par(2 * h) = \mathtt{ev}$$

This means also that $(Sign)P(Par)$ *holds. Moreover, note that* $[Sign]h := h + 1(Par)$ *does not hold since*

$$Sign(2) = Sign(3) = \mathbb{Z}^+ \text{ and } \quad \begin{array}{l} Par((\llbracket h := h+1 \rrbracket(h,2))^{\mathtt{L}}) = Par(2) = \mathtt{ev} \neq \\ Par((\llbracket h := h+1 \rrbracket(h,3))^{\mathtt{L}}) = Par(3) = \mathtt{od} \end{array}$$

This means that $\not\vdash_{\mathcal{N}_0} [Sign]P(Par)$*. On the other hand, we have that* $\vdash_{\mathcal{A}_0} (Sign)h := h + 1(Par)$ *and* $\vdash_{\mathcal{N}_0} [Par]l := 2 * h(Par)$*, therefore we can use* $R^A 6$ *since* Par *is disjunctive, and therefore we can infer* $(Sign)P(Par)$*.*

6 Discussion

In this paper we have introduced a proof system for abstract non-interference, in the general context where we are interested in understanding how data of two different groups interfere with each other. The advantage of a proof system for abstract non-interference is that checking abstract non-interference inductively on the syntax can be easily mechanized. The proof system can benefit of standard abstract interpretation methods for generating basic certificates for simple program fragments (rules **R0**). The other rules allow us to combine certificates from program fragments in a proof-theoretic certification of non-interference for programs. It is clear that our proof system is a system for certification and not for the generation of harmless attackers, since the rules are general and holds for all the abstractions satisfying the fixed restrictions. An analogous proof system, which instead allows us to generate input or output observations for abstract non-interference, can be easily derived as instantiation by fixing for each rule a possible input or output observation that we choose wrt a fixed strategy. Namely, fixed the input [output] observation we can derive a system where we choose only one output [input] abstraction in the set of all domains that satisfy the rule. Depending on the kind of application we can clearly decide the strategy for choosing such a witness and this deserves further research. Anyway, the interest in the general technology we propose in this paper is mostly related with its use in *a la* proof carrying code (PCC) verification of abstract non-interference, when mobile code is allowed. In this case in a PCC architecture, the code producer may create an abstract non-interference certificate

that attests to the fact that the code non-interference cannot be violated by the corresponding model of the observer. Then the code consumer may validate the certificate to check that the foreign code is not violating non-interference for the corresponding model of observer. The implementation of this technology requires an appropriate choice of a logic for specifying abstractions and an adequate logical framework where the logic can be manipulated. We believe that predicate abstraction [11] is a fairly simple and easily mechanizable way for reasoning about abstract domains. More appropriate logics can be designed following the ideas in [2], even though a mechanizable logic for reasoning about abstractions is currently a major challenge in this field and deserves further investigations. The language we used is quite simple. Even though abstract non-interference makes non-interference a purely semantic problem, any extension of Imp and its semantics with for example probabilistic choice, non terminating computations, and concurrency, may require a redesign of the proof systems for abstract non-interference. It would be particularly interesting to extend Imp with concurrency. In the context of language-based security, the main interest in this extension deals both with the chance to reduce protocol verification to non-interference problems and with the possibility of modeling active attackers as abstract interpretations. The models of attackers developed in abstract non-interference are indeed passive [12]. Active attackers would be particularly relevant in order to extend abstract non-interference as a language-based tool for protocol validation.

Acknowledgements

We are grateful to the anonymous referee for his useful observations and suggestions that helped us improving this paper. This paper was partially supported by the PRIN projects “SOFT” and “AIDA2”.

References

[1] G.R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2(1):56–76, 1980.

[2] A. Appel. Foundational proof-carrying code. In *Proc. of the 16th IEEE Symp. on Logic in Computer Science* (*LICS '01*), pages 247–258, Los Alamitos, Calif., 2001. IEEE Comp. Soc. Press.

[3] A. Banerjee, R. Giacobazzi, and I. Mastroeni. What you lose is what you leak: Information leakage in declassifivation policies. In *Proc. of the 23th Internat. Symp. on Mathematical Foundations of Programming Semantics* (*MFPS '07*), volume 1514 of *Electronic Notes in Theoretical Computer Science*, Amsterdam, 2007. Elsevier.

[4] D. Clark, C. Hankin, and S. Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, 2002.

[5] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating System Review*, 11(5):133–139, 1977.

[6] E. S. Cohen. Information transmission in sequential programs. In DeMillo et al., editor, *Foundations of Secure Computation*, pages 297–335, New York, 1978. Academic Press.

[7] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252, New York, 1977. ACM Press.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282, New York, 1979. ACM Press.

[10] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing: Second International Conference (SPC 2005)*, volume 3450, pages 193–209, Berlin, 2005. Springer-Verlag.

[11] C. Flanagan and S. Qadeer. Pedicate abstraction for software verification. In *Proc. of Conf. Record of the 29th ACM Symp. on Principles of Programming Languages* (*POPL '02*), pages 191–202, New York, 2002. ACM Press.

[12] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*, pages 186–197, New York, 2004. ACM-Press.

[13] R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In A. Tarlecki J. Marcinkowski, editor, *Annual Conf. of the European Association for Computer Science Logic (CSL '04)*, volume 3210, pages 280–294, Berlin, 2004. Springer-Verlag.

[14] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In S. Sagiv, editor, *Proc. of the European Symp. on Programming (ESOP '05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310, Berlin, 2005. Springer-Verlag.

[15] R. Giacobazzi and I. Mastroeni. Abstract Non-Interference. Technical report, Department of Computer Science - University of Verona, 2008.

[16] R. Giacobazzi and I. Mastroeni. Adjoining classified and unclassified information by abstract interpretation. *Journal of Computer Security*, To appear.

[17] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Los Alamitos, Calif., 1982. IEEE Comp. Soc. Press.

[18] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37:113–138, 2000.

[19] P. Laud. Semantics and program analysis of computationally secure information flow. In D. Sands, editor, *In Programming Languages and Systems, 10th European Symp. On Programming (ESOP '01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91, Berlin, 2001. Springer-Verlag.

[20] I. Mastroeni. On the rôle of abstract non-interference in language-based security. In K. Yi, editor, *Third Asian Symp. on Programming Languages and Systems (APLAS '05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433, Berlin, 2005. Springer-Verlag.

[21] G. Necula. Proof-carrying code. In *Proc. of Conf. Record of the 24th ACM Symp. on Principles of Programming Languages* (*POPL '97*), pages 106–119, New York, 1997. ACM Press.

[22] F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In D. Schmidt, editor, *Proc. of the 13th European Symp. on Programming (ESOP '04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 18–32, Berlin, 2004. Springer-Verlag.

[23] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.

[24] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[25] C. Skalka and S. Smith. Static enforcement of security with types. In *Proc. of the Fifth ACM SIGPLAN Internat. Conf. on Functional Programming(ICFP '00)*, pages 254–267, New York, 2000. ACM Press.

[26] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of The 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (*POPL '98*), pages 355–364, New York, 1998. ACM Press.

[27] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[28] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT press, Cambridge, Mass., 1993.

[29] M. Zanotti. Security typings by abstract interpretation. In M. Hermenegildo and H. Puebla, editors, *Proc. of The 9th Internat. Static Analysis Symp. (SAS '02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 360–375, Berlin, 2002. Springer-Verlag.