Property-Driven Code Obfuscations Reinterpreting Jones-Optimality in Abstract Interpretation

Roberto Giacobazzi and Isabella Mastroeni(✉)

Computer Science Department, University of Verona, Verona, Italy
{roberto.giacobazzi,isabella.mastroeni}@univr.it

Abstract. Jones-optimality determines whether a specializer improves program performances. Reinterpreting this concept in terms of the precision of an abstract interpreter means to determine whether specializing a source program is able to improve the precision of a given static analysis. In the opposite direction, a specializer failing optimality (disoptimal) would decrease the precision of the analysis when applied to the specialized code. In this paper, we exploit this reinterpretation of Jones-optimality relatively to the precision of an abstract interpreter with the aim of systematically deriving obfuscated code. In line with the idea behind Futamura's projections, we factorize the construction of the obfuscated code by separating specialization and interpretation. An interpreter specializer is then systematically made disoptimal by means of language transduction. The result is a language agnostic code obfuscator which is able to foil any given static analyzer.

Keywords: Abstract interpretation · Code obfuscation · Program interpretation · Jones-optimality

1 Introduction

Code obfuscation relies upon the idea of making security inseparable from code: *a program, or parts of it, are transformed in order to make them hard to understand or analyze* [10]. This technology is increasingly relevant in software security, providing an effective way for facing the problem of code protection against reverse engineering. This contributes to comprehensive digital asset protection, with applications in DRM systems, IPP systems, tamper resistant applications, watermarking and fingerprinting, and white-box cryptography [8,9].

Obfuscation [2] exploits, by a suitably designed program transformation, the intensional nature of program analysis [4,24], namely the fact that the precision of a program analysis algorithm depends upon the way the program is written and on how data structures are used. The attack scenario here considers the protection of a program—the asset, from an attacker which is implemented by a program analysis algorithm—the so called hostile observer.

In this paper, we consider program analysis as implemented by an abstract interpreter [13]. This is general enough to include most effective sound program

G. Singh and C. Urban (Eds.): SAS 2022, LNCS 13790, pp. 247–271, 2022.
https://doi.org/10.1007/978-3-031-22308-2_12

analysis algorithms. The abstraction here plays the role of constraining the interpreter (i.e., a Universal Turing Machine) within the boundaries of expressivity as given by the chosen abstract domain. On the one hand, this realizes a case principle in computer security, where the security of a system (e.g., an encryption protocol) is always proved relatively to a *constrained attacker*, (e.g., by computational complexity). On the other, because any effective attack on code cannot avoid some form of automation of program analysis, this model can fruitfully represent a relevant part of the action of code attack by reverse engineering.

It is known that, by transforming a code we can improve or reduce the precision of any analyzer. It is in general impossible to design a compiler that automatically removes from any program all the false alarms produced by a non straightforward abstract interpreter [4]. However it is instead always possible to inject arbitrary many false alarms by compilation. One of such obfuscating compilers can be simply designed by specializing a suitably designed (called distorted) interpreter [20]. The key observation relies upon the semantic equivalence between the source code and an interpreter specialized on this code. In this case, [26]: (1) The transformed program (resulting from the specialization process) inherits the *programming style* of the interpreter; (2) The transformed program inherits the *semantics* of the original program. The reason for (1) is that the transformed program is the result of the specialization of the code of the interpreter. The reason for (2) is that even though the transformed program may be a disguised form of the source code P, a correct interpreter must faithfully execute the operations that P specifies. It is therefore always possible to act on the intensional properties of programs, and hence on the precision of program analysis, by specializing a suitably designed interpreter [20].

Paper Contribution. In this paper we go deeper into building obfuscating compilers by considering the role of the specializer and its interplay with the given interpreter in the action of producing obfuscated code. The notion of Jones optimality [25,27] helps to give us the compass for understanding the role of program specialization in code obfuscation. Jones optimality was originally introduced to prove whether by compilation it is possible to improve program performance by removing the so called *interpretational overhead* [27]. We reinterpret Jones optimality in the light of the accuracy of an abstract interpreter. In particular, we show that obfuscating programs by specializing interpreters can be seen as a peculiar, and non-standard, case of Jones-(dis)optimality [25,27], where, instead of considering performances, we consider precision. We introduce a new notion of optimality (Sect. 5) stating that a specializer is optimal w.r.t. an abstract interpreter if the abstract interpreter is complete (viz. precise [22]) for the resulting program obtained by specializing the concrete interpreter with the source code. Of course, optimal specializers removing all false alarms cannot exist for all programs and non straightforward abstract interpreters, otherwise by the second Futamura projection such compiler would exist [4]. However the degree of optimality of the specializer shows how the specializer is able to remove the imprecision injected by a distorted interpreter. In the case of code flattening, the code obtained by specializing a vanilla interpreter with the source code produces a truly flattened code whenever the program counter is forced to be a dynamic structure [20]. This inhibits a simple specializer to reconstruct the source structure, hence forcing its disoptimal behavior.

On this basis, we derive a constructive technique for building obfuscating compilers which are driven by the property to hide. The distortion phase is built by means of suitable transducers (Sect. 3) that syntactically act on code in order to make a fixed abstract interpreter incomplete for the property to hide. The core structure of our property-driven obfuscating compilers (Sect. 6) is language and property independent. The main conceptual innovation is in the correspondence between a modified version of Jones optimality, where concrete execution time is replaced by the precision of the abstract interpreter, and the process of protecting a program from the analysis obtained by that abstract interpreter. In order to formally characterize this correspondence we need to rethink program interpretation by separating the syntactic parsing from the semantic interpretation (Sect. 4). This allows us to perform the distortion process on the syntactic phase only, without changing the semantic interpretation of code, hence further separating distortion from interpretation.

Related Works. The most related work is [20], based on the seminal paper [19], where obfuscation was formalized by means of completeness and interpreter specialization. Giacobazzi et al. [20] provide precisely the theoretical bases for obfuscating programs by interpreter specialization, in order to force intensional properties affecting the precision of a given static analysis. With respect to [20], we focus the attention on what we want to protect rather than on what the attacker can observe/analyze. Moreover, Giacobazzi et al. [20] did not provide any systematic approach for deriving the distorted interpreters. Our aim is to fill the gap between the identification of the property to make obscure and the process for building the distorted interpreter to specialize for obscuring the property.

Dalla Preda and Mastroeni [16] exploit the relation between obfuscation and completeness to design property-driven obfuscation strategies as program transformations *revealing* (preserving) some fixed semantic property while *concealing* a property to protect. In this work, it is also shown that the obfuscation approach based on distorted interpreters [20] is precisely a technique for revealing the I/O program semantics while concealing a given property to protect. The problem with this work is that it still does not provide a constructive method for obfuscating programs, but only a theoretical framework for designing obfuscation strategies. Finally, Giacobazzi et al. [21] exploit the relation between completeness and obfuscation for "measuring" obfuscation potency, namely the obfuscator capability of hiding properties.

2 Background

2.1 The Language $\mathfrak{L}$ and Control Flow Graphs

Following [5,30] (see also [37]) we consider the language $\mathfrak{L}$ of regular commands in Fig. 1 (where + denotes non-deterministic choice and $*$ is the Kleene closure),

$$\begin{array}{rl} \text{Exp} \ni \mathsf{e} ::= & \mathsf{a} \mid \mathsf{b} \\ \text{AExp} \ni \mathsf{a} ::= & x \mid n \mid \mathsf{a}+\mathsf{a} \mid \mathsf{a}-\mathsf{a} \mid \mathsf{a}*\mathsf{a} \\ \text{BExp} \ni \mathsf{b} ::= & x \mid \mathit{true} \mid \mathit{false} \mid \mathsf{e}=\mathsf{e} \mid \mathsf{e}>\mathsf{e} \mid \mathsf{e}<\mathsf{e} \mid \mathsf{b}\wedge\mathsf{b} \mid \neg\mathsf{b} \\ \mathsf{Stm} \ni \mathsf{c} ::= & \mathbf{skip}; \mid x := \mathsf{e}; \mid \mathsf{b}; \\ \mathsf{Stms} \ni \mathtt{C} ::= & \mathsf{c} \mid \mathtt{C}\,\mathtt{C} \mid {}_{+}\langle \mathtt{C} + \mathtt{C}\rangle_{+}; \mid {}_{*}\langle \mathtt{C}\rangle_{*}; \\ \mathfrak{L} \ni \mathsf{P} ::= & \{\,\mathtt{C}\,\} \text{ (programs)} \qquad \mathit{Var} \ni x \text{ (variables)}, \qquad n \in \mathbb{Z} \text{ (values)} \end{array}$$

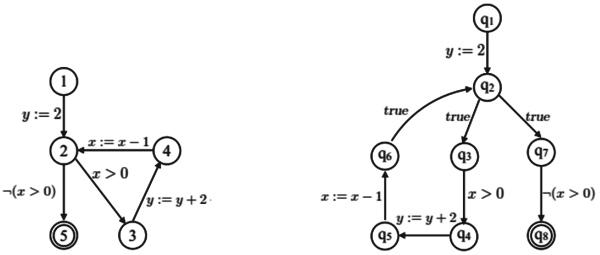
Fig. 1. Syntax of $\mathfrak{L}$

which is general enough to cover deterministic imperative languages. We complete the Bruni et al. grammar [5] with an expressions grammar, and we make some syntactic change in order to simplify the parsing and interpretation processes. In particular, we use ; not for composing statements (composition is made by concatenation of statements) but for delimiting the end of a statement. We use delimiters ${}_{+}\langle$ and $\rangle_{+}$ for determining the action range of +, we use ${}_{*}\langle$ and $\rangle_{*}$ for the range of $*$, and we use { and } for delimiting programs. Let $\mathfrak{L}$ denote also the set of programs in the language and $Var(\mathsf{P})$ the set of all the variables in $\mathsf{P} \in \mathfrak{L}$.

Programs will be graphically represented by means of their control flow graphs (CFG for short). The definition is quite standard [33], but we recall it here in order to fix the notation we use. The CFG of $\mathsf{P} \in \mathfrak{L}$ is the labeled directed graph whose nodes are program points Lab_{P} and whose edge labels are in the language $\mathrm{L}_{\mathtt{sp}} \ni \mathtt{l} ::= x := \mathsf{e} \mid \mathbf{skip} \mid \mathsf{b}$. In order to build the CFG, in the following, we will use labeled programs in $\mathfrak{L}$, namely code in $\mathfrak{L}$ where program points are labeled with values in a set of labels Lab. The labels are not in the syntax since they can be considered as program annotations added by a labeling function. Formally, let $\mathsf{P} = \{\mathtt{C}\} \in \mathfrak{L}$ its CFG is $\mathtt{Cfg}(\mathtt{C}) \overset{\text{def}}{=} Edges({}^{q_0}\mathtt{C}^{q_1}) \subseteq Lab_{\mathtt{C}} \times \mathrm{L}_{\mathtt{sp}} \times Lab_{\mathtt{C}}$, where $Edges({}^{q_0}\mathtt{C}^{q_1})$ is inductively defined on the structure of $\mathtt{C}$ (we ignore the initial and final brackets).

$$\begin{aligned} Edges({}^{q_0}\mathbf{skip};{}^{q_1}) &= \{\langle q_0, \mathbf{skip}, q_1\rangle\} \\ Edges({}^{q_0}x := \mathsf{e};{}^{q_1}) &= \{\langle q_0, x := \mathsf{e}, q_1\rangle\} \\ Edges({}^{q_0}{}_{+}\langle{}^{q_1}\mathtt{C}_1{}^{q_2} + {}^{q_3}\mathtt{C}_2{}^{q_4}\rangle_{+};{}^{q_5}) &= Edges({}^{q_1}\mathtt{C}_1{}^{q_2}) \cup Edges({}^{q_3}\mathtt{C}_2{}^{q_4}) \cup \\ &\quad \{\langle q_0, true, q_1\rangle, \langle q_0, true, q_3\rangle, \langle q_2, true, q_5\rangle, \langle q_4, true, q_5\rangle\} \\ Edges({}^{q_0}{}_{*}\langle{}^{q_1}\mathtt{C}^{q_2}\rangle_{*};{}^{q_3}) &= Edges({}^{q_1}\mathtt{C}^{q_2}) \cup \\ &\quad \{\langle q_0, true, q_1\rangle, \langle q_2, true, q_0\rangle, \langle q_0, true, q_3\rangle\} \\ Edges({}^{q_0}\mathtt{C}_1{}^{q_1}\mathtt{C}_2{}^{q_2}) &= Edges({}^{q_0}\mathtt{C}_1{}^{q_1}) \cup Edges({}^{q_1}\mathtt{C}_2{}^{q_2}) \end{aligned}$$

The nodes can be restricted to those involved in edges, i.e., $Nodes(\mathtt{Cfg}(\mathtt{C})) = \{q \mid \exists\langle q, \mathtt{l}, q'\rangle \in \mathtt{Cfg}(\mathtt{C}) \textit{ or } \langle q', \mathtt{l}, q\rangle \in \mathtt{Cfg}(\mathtt{C}),\ \mathtt{l} \in \mathrm{L}_{\mathtt{sp}}\}$. In Fig. 2 we have, on the right, an example of CFG extracted from a simple program and on the left we have a simplified version, where all the true transitions are omitted and states are relabeled.



$$\{ {}^{q_1} y := 2;\ {}^{q_2}{}_{*}\langle {}^{q_3} x > 0;\ {}^{q_4} y := y + 2;\ {}^{q_5} x := x - 1;\ {}^{q_6}\rangle_{*};\ {}^{q_7} \neg x > 0;\ {}^{q_8} \}$$

Fig. 2. Example of CFG construction.

2.2 The Language Semantics

Denotations are *memories*, i.e., partial functions $\mathbb{m} : Var \longrightarrow \mathbb{V} \cup \{\$\} \in \mathbb{M}$ where $\mathbb{V}$ is a domain of values, e.g., $\mathbb{V} \overset{\text{def}}{=} \mathbb{Z} \cup \{true, false\}$ and \$ denotes an uninstantiated value. A memory assigns values in $\mathbb{V}$ only to a finite set of variables, i.e., it is a *variable finite memory* [4]. We abuse notation by denoting as $\mathbb{M}$ precisely the set of such memories. We define $var(\mathbb{m}) \overset{\text{def}}{=} \{x \in Var \mid \mathbb{m}(x) \neq \$\}$ and for $\mathsf{M} \subseteq \mathbb{M}$, we define $var(\mathsf{M}) = \bigcup_{\mathbb{m} \in \mathsf{M}} var(\mathbb{m})$. As usual we will often represent a memory $\mathbb{m} \in \mathbb{M}$ as a tuple $[x_1/v_1, \ldots, x_n/v_n]$ of its defined variable/value pairs, i.e., such that $\mathbb{m}(y) = \$$ if $y \notin \{x_1, \ldots, x_n\}$ and $\mathbb{m}(y) = v_i$ if $y = x_i$ for all $i \in [1, n]$. Memory update is written $\mathbb{m}[x \mapsto v]$ and it associates with x the value v, while all the other associations remain the same. The concrete semantics of the program can be computed by a fine-grain small-step execution deriving the set of all the possible executions of programs. In the following, $[\![\mathsf{P}]\!] \in \wp(\mathbb{M}^*)$ denotes the set of the (terminating) program computations modeled as finite traces of memories [12], while $(\!|\mathsf{e}|\!)\,\mathbb{m}$ denotes the concrete evaluation of $\mathsf{e} \in$ Exp in the memory $\mathbb{m}$.

The Collecting Semantics. The *collecting big-step semantics* of programs in $\mathfrak{L}$ (denoted by the subscript $\mathcal{C}$) is defined as the *additive lift* of the standard I/O semantics and it is inductively defined on program's syntax. We first define the collecting semantics for $\mathsf{a} \in$ AExp, $[\![\mathsf{a}]\!]_{\mathcal{C}} : \wp(\mathbb{M}) \longrightarrow \wp(\mathbb{V})$, as additive lift to sets of memories: $[\![\mathsf{a}]\!]_{\mathcal{C}}\mathsf{M} \overset{\text{def}}{=} \{(\!|\mathsf{a}|\!)\,\mathbb{m} \mid \mathbb{m} \in \mathsf{M}\}$. Similarly, for boolean expressions $\mathsf{b} \in$ BExp, $[\![\mathsf{b}]\!]_{\mathcal{C}} : \wp(\mathbb{M}) \longrightarrow \wp(\mathbb{M})$ is defined as $[\![\mathsf{b}]\!]_{\mathcal{C}}\mathsf{M} \overset{\text{def}}{=} \{\mathbb{m} \in \mathsf{M} \mid (\!|\mathsf{b}|\!)\,\mathbb{m} = true\}$. The semantics of $\mathsf{P} = \{\mathsf{C}\}$ is $[\![\mathsf{P}]\!]_{\mathcal{C}} = [\![\mathsf{C}]\!]_{\mathcal{C}} : \wp(\mathbb{M}) \longrightarrow \wp(\mathbb{M})$ defined inductively as follows [4][1] where $[\![\mathsf{C}]\!]^1_{\mathcal{C}}\mathsf{M} \overset{\text{def}}{=} [\![\mathsf{C}]\!]_{\mathcal{C}}\mathsf{M}$ and $\forall n > 1.\ [\![\mathsf{C}]\!]^{n+1}_{\mathcal{C}}\mathsf{M} \overset{\text{def}}{=} [\![\mathsf{C}]\!]_{\mathcal{C}}\mathsf{M} \circ [\![\mathsf{C}]\!]^n_{\mathcal{C}}\mathsf{M}$:

$$\begin{aligned}
[\![\textbf{skip};]\!]_{\mathcal{C}}\mathsf{M}; &\overset{\text{def}}{=} \mathsf{M}\\
[\![x := \mathsf{e};]\!]_{\mathcal{C}}\mathsf{M} &\overset{\text{def}}{=} \mathsf{M}[x \mapsto [\![\mathsf{e}]\!]_{\mathcal{C}}\mathsf{M}] \overset{\text{def}}{=} \left\{\ \mathbb{m}[x \mapsto (\!|\mathsf{e}|\!)\,\mathbb{m}] \,\middle|\, \mathbb{m} \in \mathsf{M}\ \right\}\\
[\![{}_{+}\langle \mathsf{C}_1 + \mathsf{C}_2\rangle_{+};]\!]_{\mathcal{C}}\mathsf{M} &\overset{\text{def}}{=} [\![\mathsf{C}_1]\!]_{\mathcal{C}}\mathsf{M} \cup [\![\mathsf{C}_2]\!]_{\mathcal{C}}\mathsf{M}\\
[\![{}_{*}\langle \mathsf{C}\rangle_{*};]\!]_{\mathcal{C}}\mathsf{M} &\overset{\text{def}}{=} \bigcup\left\{\ [\![\mathsf{C}]\!]^n_{\mathcal{C}}\mathsf{M} \,\middle|\, n \in \mathbb{N}\ \right\}\\
[\![\mathsf{C}_1\mathsf{C}_2]\!]_{\mathcal{C}}\mathsf{M} &\overset{\text{def}}{=} [\![\mathsf{C}_2]\!]_{\mathcal{C}}([\![\mathsf{C}_1]\!]_{\mathcal{C}}\mathsf{M})
\end{aligned}$$

[1] We avoid labels and initial and final brackets being not used in the semantics.

Note that the collecting semantics of any non terminating program P is $[\![\mathsf{P}]\!]_{\mathcal{C}} = \varnothing$. In this case, if $\varnothing$ denotes the undefined memory, then $\lambda\mathsf{M} \subseteq \mathbb{M}.\ [\![\mathsf{P}]\!]_{\mathcal{C}}\mathsf{M}$ is the collection of memories computed by P.

In the following we will use also the notion of store, allowing us to locally denote collecting updates by associating, with each program point the collection of memories reached at each point. This allows to define a collecting small-step semantics abstracting $[\![\mathsf{P}]\!]$ [1,12]. Let us define, $\mathbb{S} \overset{\text{def}}{=} \mathit{Lab} \rightarrow \wp(\mathbb{M}) \cup \{\$\}$ such that for any $\mathbb{s} \in \mathbb{S}$ there exists a finite set of program labels $q \in \mathit{Lab}$ such that $\mathbb{s}(q) \neq \$$, in particular, given a program P we have that $\mathbb{S}_{\mathsf{P}} = \mathit{Lab}_{\mathsf{P}} \rightarrow \wp(\mathbb{M}) \cup \{\$\}$ (when not necessary or when clear from the context, we will avoid the subscript P). In the following, we will denote by $\mathbb{s}^q$ the set of memories associated with $q \in \mathit{Lab}$, i.e., $\mathbb{s}(q) \in \wp(\mathbb{M})$. For the sake of readability, we will also use the following update notation: $\mathbb{s}[q \mapsto \mathsf{M}](q') \overset{\text{def}}{=} \mathsf{M}$ if $q = q'$, $\mathbb{s}(q')$ otherwise. Moreover, we will denote by $\mathbb{s}_{\varnothing}$ the store mapping each program point to the emptyset, i.e., $\forall q.\ \mathbb{s}_{\varnothing}(q) = \varnothing$.

The Abstract Semantics. The abstract semantics of programs is an abstraction of the concrete small-step semantics [12–14], also called trace semantics. An abstract domain is a set of properties, here modeled as upper closure operators (uco for short), i.e., a monotone, extensive and idempotent operator on $\wp(\mathbb{M})$ [14]. If $\mathcal{A} \in \mathit{uco}(\wp(\mathbb{M}^*))$ is an abstraction of program traces, then we can denote by $[\![\mathsf{P}]\!]^{\mathcal{A}} \supseteq \mathcal{A}([\![\mathsf{P}]\!])$ the fix-point computation (inductively defined on the language $\mathfrak{L}$) as the $\mathcal{A}$ observation of $[\![\mathsf{P}]\!]$. In static analysis, it is quite common to define the semantic abstraction in terms of an abstraction of variable values $\mathbb{V}$. In general, if a program has n variables, then concrete values for the program are n-tuples of values. Hence, abstract domains must be parametric on the number n of variables of the program to analyze, i.e., we have to consider, as abstract domains, families of abstractions $\{\rho_n \in \mathit{uco}(\wp(\mathbb{V}^n))\}_{n\in\mathbb{N}}$ [4,15]. For the sake of readability, in the following we simply denote this family of abstraction as ρ, ignoring the technical aspect that it changes with the number of variables of the program to analyze, and we denote the corresponding abstract semantics as $[\![\mathsf{P}]\!]^{\rho}$. Given a value abstraction ρ, we can define a memory abstraction, abstracting sets of memories in $\mathbb{M}$ in abstract memories in $\mathbb{M}^{\rho}$. Define the memory abstraction as the tuple $\mathcal{A}_{\rho} = \langle \rho, \mathbb{M}^{\rho}, \alpha_{\rho}, \gamma_{\rho}\rangle$, where we define $\alpha_{\rho} : \wp(\mathbb{M}) \rightarrow \mathbb{M}^{\rho}$ ($\alpha_{\rho}(\mathsf{M}) \overset{\text{def}}{=} \lambda\langle x_1, \dots, x_n\rangle.\rho(\{\langle v_1, \dots, v_n\rangle \mid [x_1/v_1, \dots, x_n/v_n] \in \mathsf{M}\})$), while the concretization is the function $\gamma_{\rho} : \mathbb{M}^{\rho} \rightarrow \wp(\mathbb{M})$ (defined on abstract collecting memories as $\gamma_{\rho}(\overline{\mathsf{M}}) \overset{\text{def}}{=} \{[x_1/v_1, \dots, x_n/v_n] \mid \langle v_1, \dots, v_n\rangle \in \overline{\mathsf{M}}(x_1, \dots, x_n)\}$).

In order to define the abstract semantics, we define the semantics of expressions $[\![\mathsf{e}]\!]^{\rho}$ computing abstract operations in ρ, and then we define the abstract semantics of basic instructions: Let $\{x_i\}_{i\in I}$ be the set of defined variables ranging over i in the set of indexes $I = [1, n]$.

$$\begin{array}{rcl} [\![x_i := \mathsf{e}]\!]^{\rho}\overline{\mathsf{M}} & \overset{\text{def}}{=} & \lambda\langle x_1, \dots, x_n\rangle.\left\{ \langle v_1, \dots, v_i, \dots, v_n\rangle \,\middle|\, \begin{array}{l} \exists v \in \mathbb{V}.\langle v_1, \dots, v, \dots, v_n\rangle \in \\ \overline{\mathsf{M}}(x_1, \dots, x_n), v_i \in [\![\mathsf{e}]\!]^{\rho}\overline{\mathsf{M}} \end{array} \right\} \\ & \overset{\text{def}}{=} & \overline{\mathsf{M}}[x_i \mapsto [\![\mathsf{e}]\!]^{\rho}\overline{\mathsf{M}}] \\ [\![\mathbf{skip}]\!]^{\rho}\overline{\mathsf{M}} & \overset{\text{def}}{=} & \overline{\mathsf{M}} \end{array}$$

In the assignment, we consider all the tuples where the potential relation among all the variables different from x_i remains unaltered, while x_i may have any value in $[\![\mathsf{e}]\!]^{\rho}\overline{\mathsf{M}}$. The abstract semantics, of a program $\mathsf{P} = \{\mathtt{C}\} \in \mathfrak{L}$, is simply denoted as $[\![\mathsf{P}]\!]^{\rho} = [\![\mathtt{C}]\!]^{\rho} : \mathbb{M}^{\rho} \longrightarrow \mathbb{M}^{\rho}$ and it is inductively defined on the syntax of commands (loops, conditionals and compositions) as the composition of the abstract semantics of their components [4]. It is well known that abstract interpretation is not compositional, namely the composition of two best correct approximation (bca for short) semantics is not the bca semantics of the composition. This is indeed the main source of imprecision in program analysis. Note that, also for abstract semantics we can use (abstract) stores $\mathbb{S}^{\rho}$ for defining abstract collecting rules where we associate with program points abstract memories in $\mathbb{M}^{\rho}$.

Program Specialization. *Program specialization* is a source-to-source program transformation also known as *partial evaluation* [26]. A specializer is a program $\mathtt{spec}$ such that for $\mathsf{P} \in \mathfrak{L}$ with "static" input $\mathtt{s} \in D$ and "dynamic" input $\mathtt{d} \in D$ $\mathcal{S}[\mathsf{P}](\mathtt{s}, \mathtt{d}) = \mathcal{S}[\mathcal{S}[\mathtt{spec}](\mathsf{P}, \mathtt{s})]\mathtt{d}$ where $\mathcal{S}[\cdot]$ denotes generic semantics associating I/O meaning to programs independently from the language, hence distinguishing the I/O semantics from the collecting semantics $[\![\cdot]\!]_{\mathcal{C}}$. A specializer executes P in two stages: (1) P is specialized to its static input $\mathtt{s}$ yielding a "residual program" $\mathit{spec}(\mathsf{P}, \mathtt{s}) \overset{\text{def}}{=} \mathcal{S}[\mathtt{spec}](\mathsf{P}, \mathtt{s})$, (2) $\mathit{spec}(\mathsf{P}, \mathtt{s})$ can be run on P's dynamic input $\mathtt{d}$ [26].

A trivial specializer $\mathtt{spec}$ is easy to build by "freezing" the static input $\mathtt{s}$ (Kleene's *s-m-n* Theorem of the 1930s s did specialization in this way.) A number of practical program specializers exist. Published partial evaluation systems include Tempo, Ecce, Logen, Unmix, Similix and PGG [11,28,29,32,34].

3 Symbolic Finite State Machines

In this section, we define a generic notion of symbolic machine and symbolic transducer by generalizing the symbolic automata and transducers defined in the literature [17,35]. The idea consists in generalizing the *symbolic* approach (admitting potentially infinite alphabets) also to finite state machines/transducers equivalent to Turing Machines, namely with more than one stack and/or with writable input tape, while simplifying the notation, for instance by avoiding to introduce a further notion of interpretation for symbols. The following machines are non deterministic with ε transitions, where as usual ε is a special symbol used for executing transitions without reading symbols.

3.1 Finite State Machines

By finite state machines we mean any state machine with a finite number of states that reads an input sequence of symbols. Each symbol allows the execution of a transition, and final states decide which input sequences are accepted by the machine, accepted when the input reading leads to a final state. The notion is recalled only because we provide a unique parametric definition for automata and Turing machines.

Definition 1 (Finite state machines (FSM)). *A FSM is the tuple* $M = \langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Sigma, \Gamma, \mathtt{S}, \delta \rangle$ *where*

- $\mathbb{Q}$ *is a finite set of states* ($q_\iota \in \mathbb{Q}$ *initial state,* $q_{\mathtt{f}} \in \mathbb{Q}$ *final/accepting state*)[2]*;*
- Σ *and* Γ *are* finite *input and stack alphabets, respectively;*
- $\mathtt{S} \subseteq \{Stack^n\} \cup \{Input\}$ *is a set of tapes which may contain* $n \geq 0$ *Stacks (if* $n = 0$ *there are no stacks), i.e., LIFO tapes, and one Input tape, a writable and readable input tape where we can stop or move left/right*[3]*;*
- $\delta : \mathbb{Q} \times \Sigma \times \Gamma^n \to \wp(\mathbb{Q}) \times \{R, L, H\}^{\{0,1\}} \times (\Gamma^*)^n$ *is the transition function. The transition* $q \to q'$ *labeled with* $((s, M), \{t_i \to \gamma_i\}_{i\in[1,n]})$ *(read* $s \in \Sigma$ *in the state* q *with the top (popped) elements of the stacks* $\{t_i\}_{i\in[1,n]}$*, reach* q'*, push* $\{\gamma_i\}_{i\in[1,n]}$ *on the* n *stacks, and move* $M \in \{R, L, H\}$*)*[4] *iff* $\langle q', M, \{\gamma_i\}_{i\in[1,n]}\rangle \in \delta(q, s, \{t_i\}_{i\in[1,n]})$*.*

In order to make such a machine symbolic, we simply consider infinite alphabets (both for input and stack) and a recursive enumerable set of decidable predicates on the alphabet symbols. In this way, transitions are labeled with predicates allowing all the symbols satisfying the property to be read in the transition (on the input tape or on the stack).

Definition 2 (Symbolic finite state machines (SFSM)). *A SFSM is the tuple* $\langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Psi_\Sigma, \Psi_\Gamma, \mathtt{S}, \delta \rangle$ *where*

- $\mathbb{Q}$ *is a finite set of states* ($q_\iota \in \mathbb{Q}$ *initial state,* $q_{\mathtt{f}} \in \mathbb{Q}$ *final/accepting state);*
- Σ *and* Γ *are* infinite *input and stack alphabets, respectively;*
- $\Psi_\Sigma \subseteq \wp(\Sigma)$ *and* $\Psi_\Gamma \subseteq \wp(\Gamma)$ *are recursive enumerable sets of predicates on* Σ *and* Γ *(closed under logic connectives)*[5]*;*
- $\mathtt{S} \subseteq \{Stack^n\} \cup \{Input\}$ *(*$n \geq 0$ *number of stacks);*
- $\delta : \mathbb{Q} \times \Psi_\Sigma \times (\Psi_\Gamma)^n \to \wp(\mathbb{Q}) \times \{R, L, H\}^{\{0,1\}} \times (\Gamma^*)^n$*, where we have the transition* $q \to q'$ *labeled with* $((s, M), \{t_i \to \gamma_i\}_{i\in[1,n]})$ *iff* $\exists \Phi_\Sigma \in \Psi_\Sigma$*,* $\{\Phi^i_\Gamma\}_{i\in[1,n]} \subseteq \Psi_\Gamma$*,* $\gamma \in \Gamma^n$ *and* $M \in \{R, L, H\}$ *such that* $\langle q', M, \{\gamma_i\}_{i\in[1,n]}\rangle \in \delta(q, \Phi_\Sigma, \{\Phi^i_\Gamma\}_{i\in[1,n]})$*, with* $s \in \Phi_\Sigma$ *and* $\forall i \in [1, n].\, t_i \in \Phi^i_\Gamma$*.*

3.2 Finite State Transducers

Finite state transducers are finite state machine providing an output sequence of symbols for each transition. The standard generalized notion is the following.

[2] Being the machine non deterministic with ϵ-transition, w.l.g., we can suppose to have only one final state.

[3] Every FSM has a (only) readable input tape, where it is possible only to move right after each step, a finite state pushdown automaton is an automaton with also one stack, in any other cases we have a Turing Machine.

[4] Where R stands for move-right, L for move-left, and H for halt, and $\{R, L, H\}^0$ means there is no writable input tape.

[5] We avoid the interpretation function [17] simply by denoting directly the predicates extensionally, as the sets of the elements satisfying the predicate.

Definition 3 (Finite state transducers (FST)). *A finite state transducer is a FSM with an output language, i.e., a tuple* $\langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Sigma, \Gamma, \mathtt{S}, \tilde{\delta}, \Omega\rangle$, *where* $\langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Sigma, \Gamma, \mathtt{S}, \delta\rangle$ *is a FSM,* Ω *is a* finite *output alphabet, and the transition function* $\tilde{\delta} : \mathbb{Q} \times \Sigma \times \Gamma^n \rightarrow \wp(\mathbb{Q}) \times \{R, L, H\}^{\{0,1\}} \times (\Gamma^*)^n \times \Omega^*$ *is* δ *extended by returning also an output string* $\omega \in \Omega^*$ *for each input symbol read, i.e.,* $\forall (q, s, \{t_i\}_{i\in[1,n]}) \in \mathbb{Q} \times \Sigma \times \Gamma^n .\ \exists \omega \in \Omega^* .\ \tilde{\delta}(q, s, \{t_i\}_{i\in[1,n]}) \overset{\mathrm{def}}{=} \langle \delta(q, s, \{t_i\}_{i\in[1,n]}), \omega\rangle$.

In this case, we have the transition $q \rightarrow q'$ labeled with $((s/\omega, M), \{t_i \rightarrow \gamma_i\}_{i\in[1,n]})$ (read $s \in \Sigma$ in the state q with the top (popped) elements of the i-th stack $t_i \in \Gamma$ ($i \in [1, n]$) and reach state q', push $\gamma_i \in \Gamma^*$ on the i-th stack ($i \in [1, n]$), move M and provide in output the sequence $\omega \in \Omega$) iff $\exists \langle q', M, \{\gamma_i\}_{i\in[1,n]}\rangle \in \delta(q, s, \{t_i\}_{i\in[1,n]})$, and therefore $\langle q', M, \{\gamma_i\}_{i\in[1,n]}, \omega\rangle \in \tilde{\delta}(q, s, \{t_i\}_{i\in[1,n]})$.

In the symbolic extension, following the Veanes et al. [35], we simply consider a function that for each input symbol read, provides a sequence of output symbols.

Definition 4 (Symbolic finite state transducers (SFST)). *A symbolic finite state transducers is a SFSM with an output language, i.e., it is defined as a tuple* $\langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Psi_\Sigma, \Psi_\Gamma, \mathtt{S}, \tilde{\delta}, \Omega, \mathtt{f}\rangle$, *where* $\langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Psi_\Sigma, \Psi_\Gamma, \mathtt{S}, \delta\rangle$ *is a SFSM,* Ω *is an* infinite *output alphabet,* $\mathtt{f} : \Sigma \rightarrow \Omega^*$, *and the transition function* $\tilde{\delta}$ *is defined as* $\langle \delta, \mathtt{f}\rangle$.

In this case, we have the transition $q \rightarrow q'$ labeled with $((s/\mathtt{f}(s), M), \{t_i \rightarrow \gamma_i\}_{i\in[1,n]})$ (read $s \in \Sigma$ in the state q with the top (popped) elements of the i-th stack $t_i \in \Gamma$ ($i \in [1, n]$) and reach state q', push $\gamma_i \in \Gamma^*$ ($i \in [1, n]$), move M and provide in output the symbols $\mathtt{f}(s) \in \Omega^*$) iff $\exists \langle q', M, \{\gamma_i\}_{i\in[1,n]}\rangle \in \delta(q, \Phi_\Sigma, \{\Phi^i_\Gamma\}_{i\in[1,n]})$ and $s \in \Phi_\Sigma$, $\forall i \in [1, n].\ t_i \in \Phi^i_\Gamma$.

When dealing with symbolic transducers we can characterize the corresponding transduction function.

Definition 5 (Transduction). [35] *The transduction of a symbolic transducer* $\mathtt{T}$ *is the function* $\mathfrak{T}_{\mathtt{T}} : \Sigma^* \rightarrow \wp(\Omega^*)$ *where* $\mathfrak{T}_{\mathtt{T}}(\sigma) \overset{\mathrm{def}}{=} \left\{ \gamma \in \Omega^* \,\middle|\, q_\iota \xrightarrow{\sigma/\gamma} q_{\mathtt{f}} \right\}$[6].

Note that a symbolic machine $\mathtt{M} = \langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Psi_\Sigma, \Psi_\Gamma, \mathtt{S}, \delta\rangle$ can be always transformed in the transducer $\mathtt{T}_{\mathtt{M}} \overset{\mathrm{def}}{=} \langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Psi_\Sigma, \Psi_\Gamma, \mathtt{S}, \tilde{\delta}, \Sigma, \mathtt{id}\rangle$, where the output language is precisely the input one.

We can compose transducers $\mathtt{T}_1$ and $\mathtt{T}_2$ by composing their transductions [35] $\mathfrak{T}_{\mathtt{T}_1}$ and $\mathfrak{T}_{\mathtt{T}_2}$ as:

$$\mathfrak{T}_{\mathtt{T}_1} \diamond \mathfrak{T}_{\mathtt{T}_2} \overset{\mathrm{def}}{=} \lambda\sigma. \bigcup_{\gamma \in \mathfrak{T}_{\mathtt{T}_1}(\sigma)} \mathfrak{T}_{\mathtt{T}_2}(\gamma)$$

3.3 Example: Parser as *Symbolic* Pushdown Automaton

Being the language generated by a context free grammar, the parser can be modeled as a symbolic pushdown (non deterministic) automaton. In particular, it is the automaton $\mathtt{pars} \overset{\mathrm{def}}{=} \langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Psi_\Sigma, \Psi_\Gamma, \{Stack\}, \delta\rangle$, where

[6] If $\sigma = s_0 \cdot s_1 \cdots s_n$ then $q_0 \xrightarrow{\sigma/\gamma} q$ means that $q_0 \xrightarrow{s_0/\mathtt{f}(s_0)} q_1 \xrightarrow{s_1/\mathtt{f}(s_1)} \cdots \xrightarrow{s_n/\mathtt{f}(s_n)} q$, and $\gamma \overset{\mathrm{def}}{=} \mathtt{f}(s_0) \cdot \mathtt{f}(s_1) \cdots \mathtt{f}(s_n) \in \Omega^*$ where $\cdot$ stands for string concatenation.

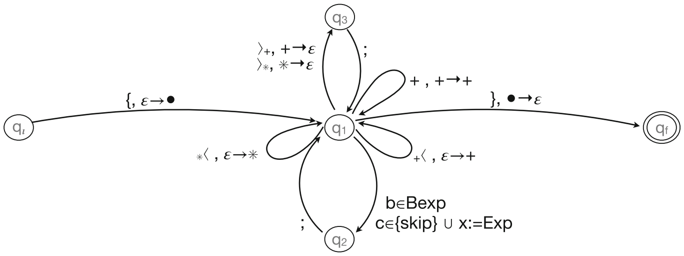

q3
$\rangle_+$, $+\to\varepsilon$
$\rangle_*$, $*\to\varepsilon$
;
+ , $+\to+$
{, $\varepsilon\to\bullet$
}, $\bullet\to\varepsilon$
q_ι
q1
qf
$_*\langle$, $\varepsilon\to*$
$_+\langle$, $\varepsilon\to+$
b∈Bexp
c∈{skip} ∪ x:=Exp
;
q2

Fig. 3. Parser

- $\mathbb{Q} \overset{\text{def}}{=} \{q_i\}_{i\in[1,3]} \cup \{q_\iota, q_{\mathtt{f}}\}$;
- $\Sigma \overset{\text{def}}{=} \left\{ x := \mathsf{e} \,\middle|\, \mathsf{e} \in \text{Exp},\ x \in Var \right\} \cup \left\{ \mathsf{b} \,\middle|\, \mathsf{b} \in \text{BExp} \right\} \cup \ \{\textbf{skip}, {}_*\langle, \rangle_*, {}_+\langle, \rangle_+, +, ;, \{, \}\}$;
- $\Psi_\Sigma \overset{\text{def}}{=} \{x := \text{Exp}, \text{BExp}\} \cup \left\{ \{s\} \,\middle|\, s \in \{{}_*\langle, \rangle_*, {}_+\langle, \rangle_+, +, ;, \textbf{skip}, \{, \}\} \right\}$;
- $\Gamma \overset{\text{def}}{=} \{+, *, \bullet\}$ and $\Psi_\Gamma = \left\{ \{t\} \,\middle|\, t \in \Gamma \right\}$[7];
- $\delta : \mathbb{Q} \times \Psi_\Sigma \times \Gamma \to \mathbb{Q} \times \Gamma^*$ is graphically defined in Fig. 3, where each transition is labeled with $(s \in \phi, t \to \gamma)$, meaning that ϕ is a predicate on Σ and $s \in \phi$, while from the stack we pop $t \in \Gamma \cup \{\varepsilon\}$ ($t = \varepsilon$ means that we don't pop anything from the stack) and we push $\gamma \in \Gamma^*$[8].

This parser simply checks brackets balance, where $+$ $(*)$ is pushed whenever a bracket is opened, and the same symbol is popped when it is closed. We can terminate only if the stack is empty (when on the top there is $\bullet$).

4 Program (Re)Interpretation

As usual the interpretation of programs is specified in two phases: The parsing phase of programs, where programs are viewed as sequences of statements, and the semantic interpretation phase, i.e., the corresponding transformation of memories/stores. The first phase is modeled as symbolic Turing Machine reading in input the sequence of symbols corresponding to the program syntax, and providing in output the precise sequences of single statements (skip and assignments) to execute and of guards to evaluate. This resulting set of sequences corresponds indeed to the CFG of the program, and on this structure we can perform the semantic interpretation phase, whose rule definitions are indeed independent from the sequence of statements/guards to execute/evaluate. Indeed, such semantic interpretation phase may be defined on concrete memories, on collecting memories of even on abstract memories, without affecting the computation of the previous interpretation phase.

[7] In this case the stack is not really symbolic.

[8] For the sake of readability we write s for singleton predicates $\{s\}$ and the empty updates of the stack, i.e., $\varepsilon \to \varepsilon$, are not depicted.

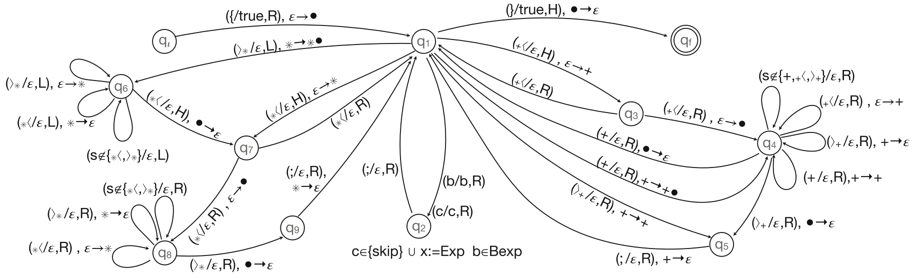


Fig. 4. Execution sequence extractor

4.1 First Phase: The Execution Sequence Extractor

The parsing of the input program, aiming also at extracting the sequence of executed statements and evaluated guards, is modeled as a symbolic Turing machine equipped with a stack. It should be clear that the input language is the language of the parser pars, that is indeed embedded in the interpreter. Hence the first component is $\mathtt{cfgEx} \overset{\text{def}}{=} \langle \mathbb{Q}, q_\iota, q_{\mathtt{f}}, \Psi_\Sigma, \Psi_\Gamma, \{Stack, Input\}, \tilde{\delta}, \Omega, \mathtt{f}\rangle$

- $\mathbb{Q} \overset{\text{def}}{=} \{q_i\}_{i\in[1,9]} \cup \{q_\iota, q_{\mathtt{f}}\}$;
- $\Sigma \overset{\text{def}}{=} \left\{\, x := \mathsf{e} \,\middle|\, \mathsf{e} \in \text{Exp},\ x \in Var \,\right\} \cup \left\{\, \mathsf{b} \,\middle|\, \mathsf{b} \in \text{BExp} \,\right\} \cup \{\mathbf{skip}, {}_*\langle, \rangle_*, {}_+\langle, \rangle_+, +, ;\}$ and $\Psi_\Sigma \overset{\text{def}}{=} \{x := \text{Exp}, \text{BExp}\} \cup \left\{\, \{s\} \,\middle|\, s \in \{{}_*\langle, \rangle_*, {}_+\langle, \rangle_+, +, ;, \mathbf{skip}, \{, \}\} \,\right\}$;
- $\Gamma \overset{\text{def}}{=} \{+, *, \bullet\}$ and $\Psi_\Gamma = \left\{\, \{s\} \,\middle|\, s \in \Gamma \,\right\}$;
- $\Omega \overset{\text{def}}{=} \mathrm{L}_{\mathtt{sp}} = \left\{\, \mathsf{b} \,\middle|\, \mathsf{b} \in \text{BExp} \,\right\} \cup \left\{\, x := \mathsf{e} \,\middle|\, \mathsf{e} \in \text{Exp},\ x \in Var \,\right\} \cup \{\mathbf{skip}\}$;
- $\delta : \mathbb{Q} \times \Psi_\Sigma \times \Gamma \rightarrow \mathbb{Q} \times \{R, L, H\} \times \Gamma^* \times \Omega$ graphically defined in Fig. 4 together with $\mathtt{f} : \Sigma \rightarrow \Omega$, where each transition is labeled, as described before, with $((s \in \phi/\mathtt{f}(s), M), t \rightarrow \gamma)$.

In particular, q_2 handles the single statement execution or the guard evaluation. q_3 handles the non deterministic choice. In particular it moves to q_1 for executing the statement on the left of + (and when it finds the symbol + it skips, in q_4, what remains up to the closed parenthesis $\rangle_+$). q_3 moves to q_4 if it wants to execute the statements on the right of +. In this case it skips all the statements up to + again in q_4. We use the stack for recognizing nested +. State q_7 handles loops, in particular it moves to q_1 for executing the body (the statements between ${}_*\langle$ and $\rangle_*$). In this case when we read $\rangle_*$ it moves to q_6 for returning back at the beginning of the loop. q_7 moves to q_8 for skipping the loop, by looking for $\rangle_*$ and continuing the execution.

In this way, if the input is a legal program, then it is accepted and the output sequences are the sequences of statements to execute and of guards to evaluate. We can observe that if we keep also the graph structure of the output (intuitively, ignoring ε-transitions and collapsing states recognizing the same language), then we obtain a graph equivalent to the CFG of the program.

Definition 6 (Partial interpreter evaluation). *The partial interpreter evaluation is the sequence/trace of statements/expressions to actually execute for*

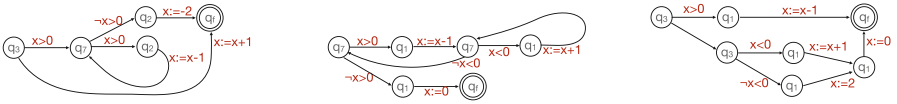


Fig. 5. Examples of interpretation

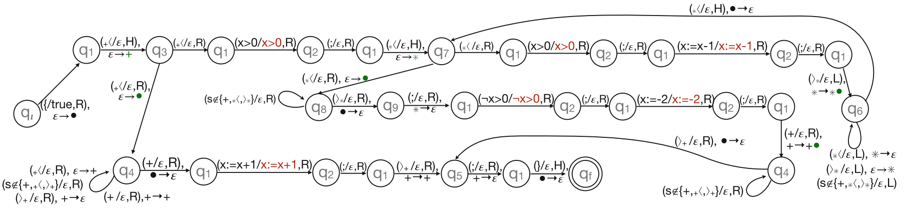
a given program. Formally, given $\mathsf{P} \in \mathfrak{L}$ *and the execution sequence extractor* $\texttt{cfgEx}$, $\langle \mathbb{Q}, q_\iota, q_{\texttt{f}}, \Psi_\Sigma, \Psi_\Gamma\{Stack, Input\}, \tilde{\delta}, \Omega, \texttt{f}\rangle$ *(*$\mathsf{P} \in \Sigma^*$*), the partial evaluation of* P *is* $\mathfrak{T}_{\texttt{cfgEx}}(\mathsf{P})$*.*

Let us denote $\texttt{cfgEx}[\mathsf{P}]$ the automaton recognizing the output language $\mathfrak{T}_{\texttt{cfgEx}}(\mathsf{P})$ of $\texttt{cfgEx}$ transduction of the input sequence P. Then we can observe that $\texttt{cfgEx}[\mathsf{P}]$ corresponds to the CFG (seen as SFSM) of P, i.e., $\texttt{cfgEx}[\mathsf{P}] \approx \texttt{Cfg}(\mathsf{P})$ (up to label renaming and minimization). Note that $\texttt{cfgEx}[\mathsf{P}]$ is a FSM (no more symbolic), (the one of the statements in P) of the infinite Σ. Let us show this correspondence on some examples.

Example 1. Consider the program P_1,

$$\{_+\langle x > 0;_*\langle x > 0; x := x - 1;\rangle_*; \neg x > 0; x := -2; \;\mathsf{+}\; \neg x > 0; x := x + 1;\rangle_+;\}$$

In the picture we depict the whole path of interpretation by means of the given interpreter: We obtain so far the automaton $\texttt{cfgEx}[\mathsf{P}_1]$ generated by the transduction of the interpreter on P_1. Each transition is labeled with $((s/o, M), t \to \gamma)$ meaning that we read s in input and we pop t from the stack, while we move M we output ω and we push γ on the stack.



If we transitively collect the transitions with output ε and we collapse states recognizing the same language, while keeping the branch and the final states, we obtain the graph on the left of Fig. 5, which corresponds to the CFG of P_1.

Consider now the program P_2

$$\{_*\langle x > 0; x := x - 1;_*\langle x < 0; x := x + 1;\rangle_*; \neg x < 0\rangle_*; \neg x > 0; x := 0;\}$$

the graph in the center of Fig. 5 corresponds to the automaton $\texttt{cfgEx}[\mathsf{P}_2]$, where the labels are the output symbols. Finally, consider the program P_3

$$\{_+\langle x > 0; x := x - 1; \;\mathsf{+}\;_+\langle x < 0; x := x + 1; \;\mathsf{+}\; \neg x < 0; x := 2;\rangle_+; x := 0;\rangle_+;\}$$

The graph on the right of Fig. 5 corresponds to $\texttt{cfgEx}[\mathsf{P}_3]$, where the labels are the output symbols.

Table 1. $\mathtt{CollR}_{\mathtt{sp}}$: The collecting interpretation rules for $\mathrm{L}_{\mathtt{sp}}$.

$$\langle\langle q_0, \textbf{skip}, q_1\rangle, \mathbb{s}\rangle \to \mathbb{s} \qquad \langle\langle q_0, x := \mathtt{e}, q_1\rangle, \mathbb{s}\rangle \to \mathbb{s}[q_1 \mapsto \mathbb{s}^{q_0}[x \mapsto [\![\mathtt{e}]\!]_{\mathcal{C}}\, \mathbb{s}^{q_0}] \cup \mathbb{s}^{q_1}]$$

$$\frac{[\![\mathtt{b}]\!]_{\mathcal{C}}\, \mathbb{s}^{q_0} \neq \varnothing}{\langle\langle q_0, \mathtt{b}, q_1\rangle, \mathbb{s}\rangle \to \mathbb{s}[q_1 \mapsto [\![\mathtt{b}]\!]_{\mathcal{C}}\, \mathbb{s}^{q_0} \cup \mathbb{s}^{q_1}]}$$

4.2 Second Phase: The Semantic Interpretation

The semantic interpretation is just an interpretation function depending on the domain of denotations, and defined for each element of $\mathrm{L}_{\mathtt{sp}}$, the output symbols to interpret. In general, given a graph $\mathtt{G}$, with initial state q_ι whose labels are in the language $\mathrm{L}_{\mathtt{sp}}$, and given a semantic rule system $\mathtt{SemR}_{\mathtt{sp}}$ defining the small-step semantics for $\mathrm{L}_{\mathtt{sp}}$, namely determining how semantic denotations in $\mathbb{D}$ (e.g., stores in $\mathbb{S}$) are transformed by the execution of elements in $\mathrm{L}_{\mathtt{sp}}$, we can interpret its paths on denotations $\mathbb{D}$ by using the following function on set of graph configurations $\mathbb{C}^{\mathtt{G}} \overset{\text{def}}{=} (\mathbb{Q} \times \mathbb{D}) \cup \mathbb{D}$, let $C \subseteq \mathbb{C}^{\mathtt{G}}$, $\mathbb{d}, \mathbb{d}' \in \mathbb{D}$ and $q_i \in \mathbb{Q}$

$$\begin{array}{l} f^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}}(q_0, \mathbb{d}) \overset{\text{def}}{=} \begin{cases} \{\mathbb{d}\} \qquad \text{(fix-point)} & \text{if } \nexists \langle q_0, \mathtt{l}, q_1\rangle \in \mathtt{G} \\ \left\{\, \langle q_1, \mathbb{d}'\rangle \,\middle|\, \exists \mathtt{l}.\ \langle\langle q_0, \mathtt{l}, q_1\rangle, \mathbb{d}\rangle \to \mathbb{d}' \in \mathtt{SemR}_{\mathtt{sp}} \,\right\} & \text{Otherwise} \end{cases} \\ f^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}}(C) \overset{\text{def}}{=} \bigcup \left\{\, f^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}}(c) \,\middle|\, c \in C \smallsetminus \mathbb{D} \,\right\} \cup \left\{ \dot{\bigcup} \left\{\, \mathbb{d} \,\middle|\, \mathbb{d} \in C \cap \mathbb{D} \,\right\} \right\} \end{array}$$

where $\dot{\bigcup}$ denotes the least upper bound on $\mathbb{D}$. Then, we can compute the fix-point interpretation of the graph $\mathtt{G}$, starting from an initial denotation $\mathbb{d}_\iota$ as the least fix-point[9] of the extensive version of $f^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}}$, i.e., $\overline{f}^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}} \overset{\text{def}}{=} \lambda C.\ f^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}}(C) \cup C$, defined in terms of the semantic rule system $\mathtt{SemR}_{\mathtt{sp}}$. This fix-point computes the set of all the reachable configurations, hence in order to extract the final/terminating ones, we have simply to consider, in this fix-point, only the configurations in $\mathbb{D}$.

$$\mathcal{S}[\mathtt{G}] \overset{\text{def}}{=} \lambda \mathbb{d}_\iota.\ (\mathit{lfp}_{\{\langle q_\iota, \mathbb{d}_\iota\rangle\}}\, \overline{f}^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}}) \cap \mathbb{D}$$

For instance, in order to define a *collecting* small-step semantics we have to define a *collecting* rule system $\mathtt{CollR}_{\mathtt{sp}}$ interpreting $\mathrm{L}_{\mathtt{sp}}$ (Table 1) on stores $\mathbb{S}$, where, given the set of initial memories M, the initial store is $\mathbb{s}_{\mathsf{M}} \in \mathbb{S}$, $\mathbb{s}_{\mathsf{M}}(q) \overset{\text{def}}{=} \mathbb{s}_{\varnothing}[q_\iota \mapsto \mathsf{M}]$, also denoted $[q_\iota \mapsto \mathsf{M}]$, Note that, the interpretation of ε is simply like interpreting *true*, and for this reason it is simply ignored.

Then the graph $\mathtt{G}$ collecting semantic interpretation is the following, where being interested only in the memories reached at the end of the program execution, we consider only the store memories at the final point $q_{\mathtt{f}}$.

$$\forall \mathsf{M} \in \wp(\mathbb{M}).\ [\![\mathtt{G}]\!]_{\mathcal{C}}\mathsf{M} \overset{\text{def}}{=} ((\mathit{lfp}_{\{\langle q_\iota, \mathbb{s}_{\mathsf{M}}\rangle\}}\, \overline{f}^{\mathtt{G}}_{\mathtt{CollR}_{\mathtt{sp}}}) \cap \mathbb{S})(q_{\mathtt{f}}).$$

[9] Given an extensive and monotone function f, its least fix-point computation starting from x is $\mathit{lfp}_x\, f \overset{\text{def}}{=} \bigvee_{n \in \mathbb{N}} f^n(x)$, where $f^0(x) = x$ and $f^{n+1}(x) = f \circ f^n(x)$.

As observed before, when we consider an abstract semantics $\mathcal{A}_\rho$[10] we obtain an abstract interpreter. The idea is simple, we can define a rule system $\mathtt{AbsR}^{\rho}_{\mathtt{sp}}$ which is precisely $\mathtt{CollR}_{\mathtt{sp}}$ where the abstract semantics $[\![\cdot]\!]^{\rho}$ for interpreting expressions is used instead of $[\![\cdot]\!]_{\mathcal{C}}$, and in terms of which we obtain, as before, a corresponding interpretation function $\overline{f}^{\mathtt{G}}_{\mathtt{AbsR}^{\rho}_{\mathtt{sp}}}$. Then we can define

$$\forall \overline{\mathsf{M}} \in \wp(\mathbb{M}^{\rho}).\ [\![\mathtt{G}]\!]^{\rho}\overline{\mathsf{M}} \overset{\text{def}}{=} ((\mathit{lfp}_{\{\langle q_\iota\,,\, \mathbb{s}_{\overline{\mathsf{M}}}\rangle\}}\ \overline{f}^{\mathtt{G}}_{\mathtt{AbsR}^{\rho}_{\mathtt{sp}}}) \cap \mathbb{S}^{\rho})(q_{\mathtt{f}})$$

4.3 Interpreting Programs

Finally we can compose the two phases and obtain the characterization of interpretation for programs $\mathsf{P} \in \mathfrak{L}$. In particular, as observed in the previous section, $\mathtt{cfgEx}[\mathsf{P}]$ returns precisely a graph with $\mathrm{L}_{\mathtt{sp}}$ as label's language, hence we can use the above semantic interpretation on this resulting graph.

Definition 7 (Program interpreter). *Given a semantic rule system* $\mathtt{SemR}_{\mathtt{sp}}$ *for* $\mathrm{L}_{\mathtt{sp}}$ *and* $\mathcal{S}[\mathtt{G}] \overset{\text{def}}{=} \lambda \mathbb{d}_\iota.\ (\mathit{lfp}_{\{\langle q_\iota\,,\, \mathbb{d}_\iota\rangle\}}\ \overline{f}^{\mathtt{G}}_{\mathtt{SemR}_{\mathtt{sp}}}) \cap \mathbb{D}$ *inductively defined on its labels in* $\mathrm{L}_{\mathtt{sp}}$. *A program interpreter for the language* $\mathfrak{L}$ *is the pair* $\mathtt{int} \overset{\text{def}}{=} \langle \mathtt{cfgEx}, \mathcal{S}[\cdot]\rangle$. *Hence,* $\forall \mathsf{P} \in \mathfrak{L}$ *the program interpretation is* $\mathcal{S}[\mathsf{P}] = \mathcal{S}[\mathtt{int}[\mathsf{P}]] \overset{\text{def}}{=} \mathcal{S}[\mathtt{cfgEx}[\mathsf{P}]]$.

For instance, the collecting interpreter for $\mathfrak{L}$ is $\langle \mathtt{cfgEx}, [\![\cdot]\!]_{\mathcal{C}}\rangle$, while an abstract interpreter w.r.t. the variable values abstraction ρ is $\langle \mathtt{cfgEx}, [\![\cdot]\!]^{\rho}\rangle$.

Combining all together, given $\mathsf{P} = \{\mathtt{C}\} \in \mathfrak{L}$, its collecting semantics starting from initial memories $\mathsf{M} \in \mathbb{M}$ is computed as follows. The following result holds by construction and by the intuitive equivalence between the collecting program semantics and the collecting interpretation of its CFG.

Proposition 1. *Let* $\mathsf{P} = \{\mathtt{C}\} \in \mathfrak{L}$, *then we have* $[\![\mathsf{P}]\!]_{\mathcal{C}} = [\![\mathtt{int}[\mathsf{P}]]\!]_{\mathcal{C}}$, *where by construction* $\forall \mathsf{M} \in \mathbb{M}.\ [\![\mathtt{int}[\mathsf{P}]]\!]_{\mathcal{C}}\mathsf{M} = ((\mathit{lfp}_{\{\langle q_\iota\,,\, \mathbb{s}_{\mathsf{M}}\rangle\}}\ \overline{f}^{\mathtt{cfgEx}[\mathsf{P}]}_{\mathtt{CollR}_{\mathtt{sp}}}) \cap \mathbb{S})(q_{\mathtt{f}})$. *In the abstract case,* $[\![\mathsf{P}]\!]^{\rho} = [\![\mathtt{int}[\mathsf{P}]]\!]^{\rho}$, *where* $\forall \overline{\mathsf{M}} \in \mathbb{M}^{\rho}.\ [\![\mathtt{int}[\mathsf{P}]]\!]^{\rho}\overline{\mathsf{M}} \overset{\text{def}}{=} ((\mathit{lfp}_{\{\langle q_\iota\,,\, \mathbb{s}_{\overline{\mathsf{M}}}\rangle\}}\ \overline{f}^{\mathtt{cfgEx}[\mathsf{P}]}_{\mathtt{AbsR}^{\rho}_{\mathtt{sp}}}) \cap \mathbb{S}^{\rho})(q_{\mathtt{f}})$.

4.4 Specializing Interpreters

In classical computational theory [31] the interpreter is indeed a program with two inputs, a fragment of code to execute and the set of initial memories from which to start execution. Our model of program interpretation distinguishes precisely between the application to the first input (the program) and the second input (the initial memories). In particular, the first phase consists precisely in applying the interpreter to the program, and the second phase consists in applying the resulting structure to the set of initial memories. In other words, it should be clear that the simple transduction $\mathtt{cfgEx}[\mathsf{P}]$ is precisely the specialization of the interpreter on the program, precomputing the interpretation computation

[10] In this case, we consider directly the semantic abstraction induced by a memory abstraction ρ since we are abstracting in the semantic interpretation phase.

involving only the code, namely the characterization of the sequences of statements to execute and of guards to evaluate. This corresponds precisely to design the program CFG. The only interpretation steps that remain to perform are those concerning the semantic interpretation, depending also on data (formally on initial memories). Hence, we can write

$$\mathit{spec}(\mathtt{int}, \mathsf{P}) = \mathit{spec}(\langle \mathtt{cfgEx}, [\![\cdot]\!]\rangle, \mathsf{P}) \overset{\text{def}}{=} \mathtt{cfgEx}[\mathsf{P}]$$

Which is indeed a specializer because by construction we have:

$$\forall \mathsf{M} \in \wp(\mathbb{M}).\ [\![\mathit{spec}(\mathtt{int}, \mathsf{P})]\!]\mathsf{M} = [\![\mathtt{cfgEx}[\mathsf{P}]]\!]\mathsf{M} = [\![\mathsf{P}]\!]\mathsf{M}.$$

5 Specializer (Dis)Optimality

In this section, we formally introduce *specializer optimality*, i.e., the specializer property characterizing the analysis precision.

5.1 Abstract Jones Optimality and Completeness

We consider code specialization in the specific context of the specialization of interpreters and abstract interpretation. Let $\mathcal{A}$ be a semantic abstraction and let `int` be an interpreter. Note that, given the new definition of interpreter, a semantic abstraction could be both an approximation of the CFG, providing a CFG containing the concrete computations and/or an abstraction of data manipulated by programs. The following definition reinterprets the notion of Jones optimality where computation time is replaced by the precision of an abstract interpreter.

Definition 8 ($\mathcal{A}$-optimality). *A specializer* `spec`*, implementing the function spec, is $\mathcal{A}$-optimal w.r.t the interpreter* `int` *if it does not lose precision w.r.t. the semantic abstraction $\mathcal{A}$, i.e., if $[\![\mathit{spec}(\mathtt{int}, \mathsf{P})]\!]^{\mathcal{A}} \sqsubseteq [\![\mathsf{P}]\!]^{\mathcal{A}}$.*

Note that, if we replace $[\![P]\!]^{\mathcal{A}}$ with time complexity of P, this definition boils down precisely to Jones-optimality [25,27]. When applied to the case of abstract interpretation, a stronger property is also important, as specified in the following definition.

Definition 9 ($\mathcal{A}$-suboptimality). *A specializer* `spec` *implementing the function spec is $\mathcal{A}$-suboptimal w.r.t. an interpreter* `int` *if it does not add precision, i.e., $\forall \mathsf{P} \in \mathfrak{L}\,.\ [\![\mathit{spec}(\mathtt{int}, \mathsf{P})]\!]^{\mathcal{A}} = [\![\mathsf{P}]\!]^{\mathcal{A}}$.*

While for straightforward abstractions $\mathcal{A}$, such as the identity and the top abstraction, $\mathcal{A}$-suboptimality always holds, for non-straightforward abstractions $\mathcal{A}$, $\mathcal{A}$-suboptimality depends upon the specializer `spec` and the interpreter `int`.

Proposition 2. *Given a self-interpreter (written in the interpreted language)* `int`*, there exists an $\mathcal{A}$-(sub)optimal specializer.*

Proof. The idea is similar to the case of trivial Jones-optimal [25]. Being $\mathtt{int}$ a self interpreter, there exists a trivial $\mathcal{A}$-optimal specializer semantics $\overline{spec}$, i.e.,

$$\overline{spec}(\mathsf{P}, x) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } \mathsf{P} = \mathtt{int} \\ spec(\mathsf{P}, x) & \text{otherwise} \end{cases}$$

which is a (computable) specializer semantics, since $[\![\overline{spec}(\mathtt{int}, \mathsf{P})]\!]^{\mathcal{A}} = [\![\mathsf{P}]\!]^{\mathcal{A}}$.

Note that, when $\mathtt{spec}$ is not $\mathcal{A}$-optimal, it may happen that we don't have any relation between the original program and the specialized one, or it may happen that the specialized program is indeed less precise. In this case, namely when $\exists \mathsf{P} \in \mathfrak{L}, \exists \mathsf{M} \subseteq \mathbb{M}.\ [\![spec(\mathtt{int}, \mathsf{P})]\!]^{\mathcal{A}}\mathsf{M} \sqsupset [\![\mathsf{P}]\!]^{\mathcal{A}}\mathsf{M}$, then we say that $\mathtt{spec}$, implementing the function *spec*, is $\mathcal{A}$-*disoptimal* w.r.t. the interpreter $\mathtt{int}$. Note that, both the notions of optimality and disoptimality may happen on a specific program, in particular we can say that $\mathtt{spec}$ is ρ-suboptimal/optimal/disoptimal w.r.t. the interpreter $\mathtt{int}$ for the program P if the corresponding definition holds for P, i.e., $[\![spec(\mathtt{int}, \mathsf{P})]\!]^{\mathcal{A}} = [\![\mathsf{P}]\!]^{\mathcal{A}}$(resp. $\sqsubseteq$ or $\sqsupset$).

Let us show the relation of optimality with precision in the abstract analysis, namely w.r.t completeness. Completeness in abstract interpretation means that the abstract computation $[\![\cdot]\!]^{\mathcal{A}}$ is precise as the abstraction of the concrete computation, i.e., $\forall \mathsf{P} \in \mathfrak{L}\,.[\![\mathsf{P}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}]\!])$ [14,22]. In this case we say that $\mathcal{A}$ is complete, while, if it holds for a program P, we say that $\mathcal{A}$ is complete for P.

Lemma 1. *Let* $\mathtt{spec}$ *be a (concrete) specializer implementing spec,* $\mathtt{int}$ *a collecting self interpreter and* $\mathcal{A}$ *a semantic abstraction. Let* $\forall \mathsf{P}.\ \mathsf{P}^s_{\mathtt{int}} \stackrel{\text{def}}{=} spec(\mathtt{int}, \mathsf{P})$*, then we have the following facts:*

1. $\mathtt{spec}$ $\mathcal{A}$*-suboptimal w.r.t.* $\mathtt{int} \Rightarrow \forall \mathsf{P}.$ *(*$\mathcal{A}$ *complete for* $\mathsf{P} \Leftrightarrow \mathcal{A}$ *complete for* $\mathsf{P}^s_{\mathtt{int}}$*);*
2. $\forall \mathsf{P}.$ *(*$\mathcal{A}$ *complete for* P *and* $\mathsf{P}^s_{\mathtt{int}} \Rightarrow \mathtt{spec}$ $\mathcal{A}$*-suboptimal w.r.t.* $\mathtt{int}$ *and* P*);*
3. $\mathtt{spec}$ $\mathcal{A}$*-optimal w.r.t.* $\mathtt{int} \Rightarrow \forall \mathsf{P}.$ *(*$\mathcal{A}$ *complete for* $\mathsf{P} \Rightarrow \mathcal{A}$ *complete for* $\mathsf{P}^s_{\mathtt{int}}$*);*
4. $\forall \mathsf{P}.$ *(*$\mathcal{A}$ *complete for* $\mathsf{P}^s_{\mathtt{int}} \Rightarrow \mathtt{spec}$ $\mathcal{A}$*-optimal w.r.t.* $\mathtt{int}$ *and* P*).*

Proof. Let us recall that, by construction $[\![\mathsf{P}]\!]_{\mathcal{C}} = [\![\mathsf{P}^s_{\mathtt{int}}]\!]_{\mathcal{C}}$.

1. If $\mathtt{spec}$ is $\mathcal{A}$-suboptimal then $\forall \mathsf{P}.\ [\![\mathsf{P}]\!]^{\mathcal{A}} = [\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}}$. Suppose $\mathcal{A}$ complete for P, then it means that $[\![\mathsf{P}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}]\!])$, therefore we have $[\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}} = [\![\mathsf{P}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}]\!]_{\mathcal{C}}) = \mathcal{A}([\![\mathsf{P}^s_{\mathtt{int}}]\!]_{\mathcal{C}})$ hence we have completeness also for $\mathsf{P}^s_{\mathtt{int}}$ Analogously we can prove completeness for $\mathsf{P}$when we have completeness for $\mathsf{P}^s_{\mathtt{int}}$ Intuitively, we have not the inverse implication since when $\mathcal{A}$ is both incomplete for P and $\mathsf{P}^s_{\mathtt{int}}$we cannot imply anything on the optimality of the specializer, while when it is complete for both we can prove the following result.
2. If $\mathcal{A}$is complete for both P and $\mathsf{P}^s_{\mathtt{int}}$then $[\![\mathsf{P}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}]\!])$and $[\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}^s_{\mathtt{int}}]\!]_{\mathcal{C}})$ Therefore $[\![\mathsf{P}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}]\!]) = \mathcal{A}([\![\mathsf{P}^s_{\mathtt{int}}]\!]) = [\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}}$ meaning suboptimality w.r.t. $\mathtt{int}$and P
3. If $\mathtt{spec}$ is $\mathcal{A}$-optimal then $\forall \mathsf{P}.\ [\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}} \sqsubseteq [\![\mathsf{P}]\!]^{\mathcal{A}}$ Suppose $\mathcal{A}$ complete for P, then it means that $[\![\mathsf{P}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}]\!])$ therefore we have $\mathcal{A}([\![\mathsf{P}^s_{\mathtt{int}}]\!]) \sqsubseteq [\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}} \sqsubseteq [\![\mathsf{P}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}]\!]) = \mathcal{A}([\![\mathsf{P}^s_{\mathtt{int}}]\!])$ Hence they are all equalities, and therefore we have completeness of $\mathsf{P}^s_{\mathtt{int}}$

4. If, given $\mathsf{P} \in \mathfrak{L}$, we have $\mathcal{A}$ complete for $\mathsf{P}^s_{\mathtt{int}}$then $[\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}^s_{\mathtt{int}}]\!])$ hence we have $\forall \mathsf{P}.\ [\![\mathsf{P}^s_{\mathtt{int}}]\!]^{\mathcal{A}} = \mathcal{A}([\![\mathsf{P}^s_{\mathtt{int}}]\!]) = \mathcal{A}([\![\mathsf{P}]\!]) \sqsubseteq [\![\mathsf{P}]\!]^{\mathcal{A}}$.

Theorem 1. *Let* spec *be a (concrete) specializer implementing spec,* int *a collecting interpreter and $\mathcal{A}$ a semantic abstraction. Let $\forall \mathsf{P}.\ \mathsf{P}^s_{\mathtt{int}} \overset{\text{def}}{=} spec(\mathtt{int}, \mathsf{P})$, then we have that $\forall \mathsf{P} \in \mathfrak{L}$ if $\mathcal{A}$ complete for* P *then: spec $\mathcal{A}$-optimal w.r.t.* int *and* P $\Leftrightarrow$ *$\mathcal{A}$ complete for $\mathsf{P}^s_{\mathtt{int}}$ $\Leftrightarrow$ spec $\mathcal{A}$-suboptimal w.r.t.* int *and* P.

Proof. Trivial by Lemma 1. In particular, if *spec* is $\mathcal{A}$-optimal w.r.t. int and P, then by the hypothesis of completeness on P and by Lemma 1(3) we have $\mathcal{A}$ is complete for $\mathsf{P}^s_{\mathtt{int}}$, and by Lemma 1(4) we have *spec* $\mathcal{A}$-suboptimal w.r.t. int and P. Finally, by definition *spec* $\mathcal{A}$-suboptimal is also $\mathcal{A}$-optimal.

These results tell us that in order to obtain an obfuscator by specializing an interpreter we have to use a specializer which is not optimal w.r.t. the observation and the interpreter.

We now formalize the relation between the specializer optimality, w.r.t., a given interpreter, and our capability of obfuscating the source program by specializing an interpreter. Our aim is to exploit this relation for driving the distortion of the interpreter by means of the property we have to obfuscate, namely by the property to hide from the analyzer.

Theorem 2. *Given an interpreter* int *and a specializer* spec *(with semantics spec), we define the program transformer $\mathfrak{O}(\mathsf{P}) \overset{\text{def}}{=} spec(\mathtt{int}, \mathsf{P})$. For any program* P*, $\mathfrak{O}(\mathsf{P})$ is an obfuscation of* P *w.r.t. the semantic abstraction $\mathcal{A}$ iff spec is $\mathcal{A}$-disoptimal w.r.t.* int *on* P.

Proof. If *spec* is $\mathcal{A}$-disoptimal w.r.t. int then $[\![\mathsf{P}]\!]^{\mathcal{A}} \sqsubset [\![spec(\mathtt{int}, \mathsf{P})]\!]^{\mathcal{A}}$, and therefore we have $[\![\mathsf{P}]\!]^{\mathcal{A}} \sqsubset [\![\mathfrak{O}(\mathsf{P})]\!]^{\mathcal{A}}$ by definition of $\mathfrak{O}(\mathsf{P})$, meaning that $\mathfrak{O}(\mathsf{P})$ is an obsfuscator for what we observed in previous sections. On the other hand, if $[\![\mathsf{P}]\!]^{\mathcal{A}} \sqsubset [\![\mathfrak{O}(\mathsf{P})]\!]^{\mathcal{A}}$, then surely we have $[\![\mathsf{P}]\!]^{\mathcal{A}} \sqsubset [\![spec(\mathtt{int}, \mathsf{P})]\!]^{\mathcal{A}}$ (by definition of $\mathfrak{O}(\mathsf{P})$) meaning that the specializer is $\mathcal{A}$-disoptimal.

At this point it should be clear why disoptimality is defined by keeping a (strict) approximation relation and not by losing any relation. Indeed, in this way it guarantees the semantics observed on the obfuscated program to be conservative w.r.t. the original semantics by containing it. In this way, while we force to lose the property we aim at obfuscating, we also partially keep the semantics of the original program by over approximating it.

Finally, since $\mathcal{A}$-disoptimality depends on both the specializer and the interpreter, we can choose to force disoptimality by leaving the specializer unchanged, and by acting only on the interpreter. We characterize a distortion of the interpreter able to make the specializer $\mathcal{A}$-disoptimal. This idea is enforced by two aspects: (1) Due to the construction proposed, the distortion can be simply characterized as a parser inductively transforming the code in a semantically equivalent program[11]; (2) Distortion can be driven by the notion of completeness, which is a semantic property strongly related to optimality as shown before.

[11] Note that semantic equivalence on single (atomic) statements is decidable.

5.2 Distorting Interpreters

In order to understand how to distort interpreters for inducing incompleteness we observe the following facts:

– For each (not straightforward) abstraction there always exists a program/code whose abstract semantics is incomplete, hence we can always characterize syntactic elements making an abstract semantics incomplete [23];
– We can think of transforming statements that will be executed in a semantic preserving way, yet including these elements;
– Clearly, it is not decidable to determine which statements will be executed, therefore in order to introduce incomplete syntactic elements in our code that will be surely executed, in general, we need to transform all program statements;
– A language parser transforming each statements by introducing incomplete elements for a fixed abstraction $\mathcal{A}$ produces the required results.
– By composing this parser with the interpreter, we obtain a distorted interpreter for which the control flow graph interpretation, i.e., its specialization w.r.t. the input program, is disoptimal, therefore obtaining an obfuscator obscuring the property expressed by $\mathcal{A}$ on any input program.

We define a module that transforms the interpreter as a SFST accepting in input the language of the parser and in output the required syntactic transformation. This transducer is then composed with the interpreter, forcing the distorter output language to be the input one of the interpreter, hence generating a transformed CFG. This transformed CFG will be the source of interpretation.

Definition 10 (Interpreter Distorter). *Let* $\mathtt{int} = \langle \mathtt{cfgEx}, \mathcal{S}[\cdot] \rangle$ *be an interpreter accepting in input the language* $\mathfrak{L}$*. An interpreter distorter* $\mathtt{D}$ *is a SFST whose output language is a (strict) subset of* $\mathfrak{L}$ *and preserving program semantics, i.e.,* $\mathtt{int}^{\mathtt{D}} \overset{\mathrm{def}}{=} \langle \mathtt{D} \diamond \mathtt{cfgEx}, \mathcal{S}[\cdot] \rangle$ *is the distorted interpreter if* $\forall \mathsf{P} \in \mathfrak{L}\,.\,\mathfrak{T}_{\mathtt{D}}(\mathsf{P}) \in \mathfrak{L}$ *and* $\mathcal{S}[\mathsf{P}] = \mathcal{S}[\mathfrak{T}_{\mathtt{D}}(\mathsf{P})]$*.*

Note that, if $\mathtt{T}_{\mathtt{pars}}$ is the trivial transducer associated with $\mathtt{pars}$, then $\mathtt{T}_{\mathtt{pars}} \diamond \mathtt{cfgEx} = \mathtt{cfgEx}$.

Example: Trivial Syntactic Distorter. Suppose $\mathtt{f}_{\mathsf{b}} : \mathrm{BExp} \to \mathrm{BExp}$ and $\mathtt{f}_{\mathsf{c}} : \mathsf{Stm} \to \mathsf{Stm}$ be semantic preserving transformers, i.e., $\forall \mathsf{b} \in \mathrm{BExp}\,.\,[\![\mathsf{b}]\!] = [\![\mathtt{f}_{\mathsf{b}}(\mathsf{b})]\!]$ and $\forall \mathsf{c} \in \mathsf{Stm}.\ [\![\mathsf{c}]\!] = [\![\mathtt{f}_{\mathsf{c}}(\mathsf{c})]\!]$, then the distorter in Fig. 6 is a trivial distorter, where the empty stack update $\varepsilon \to \varepsilon$ and the R moves are omitted in the transitions. As the parser, it is a symbolic pushdown automaton, which transforms the code while parsing it.

Theorem 3. *Let* $\mathtt{int}$ *be a* $\mathfrak{L}$ *interpreter and let* $\mathtt{int}^{\mathtt{D}}$ *be distorted by* $\mathtt{D}$*. Then* $\forall \mathsf{P} = \{\mathsf{C} \in\} \,\mathfrak{L}$ *we have* $[\![\mathtt{int}[\mathsf{P}]]\!] = [\![\mathtt{int}^{\mathtt{D}}[\mathsf{P}]]\!]$*.*

This result tells us that the semantics of the program obtained by specializing the distorted interpreter is the same as the semantics of the original program, providing so far a potential obfuscation. Whether it is an obfuscation depends on the property we want to hide, and therefore it depends on the specific distortion and on the specific program semantics.

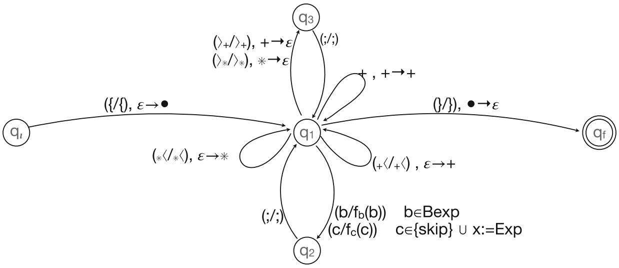


Fig. 6. The trivial distortion

6 Obfuscation by Specializing Distorted Interpreters

In this section we show some examples of interpreter distortion making the CFG specializer disoptimal, therefore producing obfuscated code. As observed before, the first step is to fix the property to hide by obfuscation and a syntactic element making the analysis of this property incomplete. In other words, given a program property $\mathcal{A}$ to hide, we define a $\mathcal{A}$-distorting interpreter, namely an interpreter for which the specializer implementing *spec* is $\mathcal{A}$-disoptimal. This is obtained by isolating a syntactic object producing imprecision and by embedding these objects into the code in such a way the abstract interpretation on $\mathcal{A}$ of the resulting program becomes incomplete.

As an example, in order to obfuscate the program control flow we observe that incompleteness is obtained by injecting opaque predicates in the program [20], hence by using the trivial interpreter distorter we can obfuscate CFG simply by defining the following transformers where $\underline{\mathsf{b}} \in \mathrm{BExp}$ is an opaque predicate (e.g., an always true predicate), $\mathsf{b}' \in \mathrm{BExp}$ and $\mathsf{c}' \in \mathsf{Stms}$.[12]

$$\begin{aligned}\mathtt{f}_{\mathsf{c}}(\mathsf{c}) &\overset{\text{def}}{=} {}_{+}\langle \underline{\mathsf{b}}; \mathsf{c} + \neg\underline{\mathsf{b}}; \mathsf{c}'\rangle_{+}\\ \mathtt{f}_{\mathsf{b}}(\mathsf{b}) &\overset{\text{def}}{=} {}_{+}\langle \underline{\mathsf{b}}; \mathsf{b} + \neg\underline{\mathsf{b}}; \mathsf{b}'\rangle_{+}\end{aligned}$$

Data Obfuscation: Parity Obfuscation. Suppose we aim at obfuscating the parity *Par* (formalized as the well-known parity abstraction on numerical values) observation of data [18], i.e., of variable integer values[13]. First of all we provide a general data abstraction distorter, where we define an expression transformers ($\mathtt{f}_{\mathsf{c}}$ and $\mathtt{f}_{\mathsf{b}}$) executing a value transformation hiding the property to obfuscate when storing data, and a complementary (i.e., which composed returns the identity) transformation when accessing data [18]. For instance, for obfuscating parity, when we store data we can multiply by 2 (hiding parity),

[12] For space reasons, and being quite intuitive, we do not provide the whole formalization of obfuscation by opaque predicates.

[13] For the sake of simplicity we suppose, without losing generality, that variables can only contain integer values, not boolean.

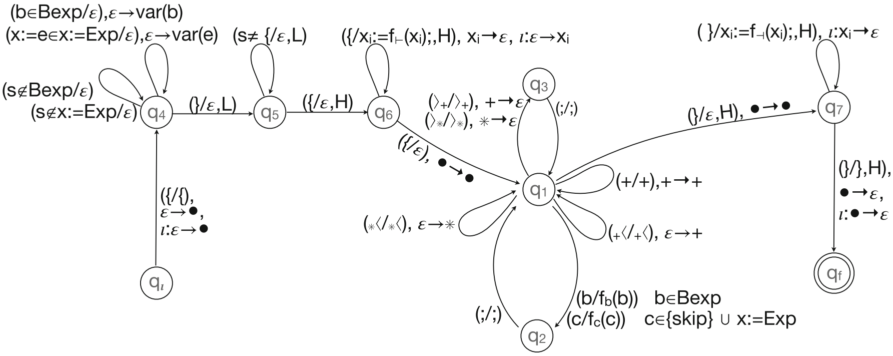


Fig. 7. Data obfuscation interpreter distorter $\mathtt{D}_{\mathtt{Data}}$.

and therefore when we access data we have to divide by 2. In order to make everything work, we have to make analogous transformations of variables values at the beginning ($\mathtt{f}_{\vdash}$) and at the end of the program ($\mathtt{f}_{\dashv}$). Hence, a simple pushdown automaton is not sufficient since, for these last variables updates, we have to scan the whole program for collecting variable names. In particular, in Fig. 7 we define $\mathtt{D}_{\mathtt{Data}}$, where q_4 extracts all the variables accessed in the program and put them on a stack ($var(\mathsf{e})$ and $var(\mathsf{b})$ are sequences of identifiers), q_5 goes back to beginning of the program and q_6 adds, at the beginning, an assignment involving each variable extracted to which we assign its transformation by $\mathtt{f}_{\vdash}$. While creating this assignments we keep in the stack ι the names of all the variables for creating the final assignments at the end of the program (state q_7) assigning to each variable its transformation by $\mathtt{f}_{\dashv}$.

Hence, we define $\mathtt{D}_{Par}$ by defining the transformations $\mathtt{f}_{\mathtt{c}}$, $\mathtt{f}_{\mathtt{b}}$, $\mathtt{f}_{\vdash}$ and $\mathtt{f}_{\dashv}$ as follows, where $Var(\mathsf{P}) = \{x_i\}_{i\in[1,n]}$

$$\begin{array}{rclrcl}
\mathtt{f}_{\vdash}(x) & \overset{\text{def}}{=} & 2 * x & \mathtt{f}_{\dashv}(x) & \overset{\text{def}}{=} & x/2 \\
\mathtt{f}_{\mathtt{c}}(\mathsf{c}) & \overset{\text{def}}{=} & \begin{cases} 2 * \mathtt{f}_{\mathtt{ex}}(\mathsf{a}) & \text{if } \mathsf{c} = x := \mathsf{a} \\ \textbf{skip} & \text{if } \mathsf{c} = \textbf{skip} \end{cases} & \mathtt{f}_{\mathtt{b}}(\mathsf{b}) & \overset{\text{def}}{=} & \mathtt{f}_{\mathtt{ex}}(\mathsf{b}) \\
\mathtt{f}_{\mathtt{ex}}(x) & \overset{\text{def}}{=} & x/2 & \mathtt{f}_{\mathtt{ex}}(n) & \overset{\text{def}}{=} & n \\
\mathtt{f}_{\mathtt{ex}}(\mathsf{e}\ bop\ \mathsf{e}) & \overset{\text{def}}{=} & \mathtt{f}_{\mathtt{ex}}(\mathsf{e})\ bop\ \mathtt{f}_{\mathtt{ex}}(\mathsf{e}) & \mathtt{f}_{\mathtt{ex}}(\neg\mathsf{b}) & \overset{\text{def}}{=} & \neg\mathtt{f}_{\mathtt{ex}}(\mathsf{b})
\end{array}$$

In Fig. 8, on the left, we have the parity obfuscation of the program whose CFG is depicted in Fig. 2. The following theorem tells us that the so far designed interpreter distorter provides us with a parity obfuscation technique by specializing the distorted interpreter. Intuitively, at the first access to each variable, by dividing by 2 its value, we lose its parity, adding analysis imprecision and making so far the abstract parity interpreter incomplete, the specializer disoptimal and the resulting program obfuscated w.r.t. parity.

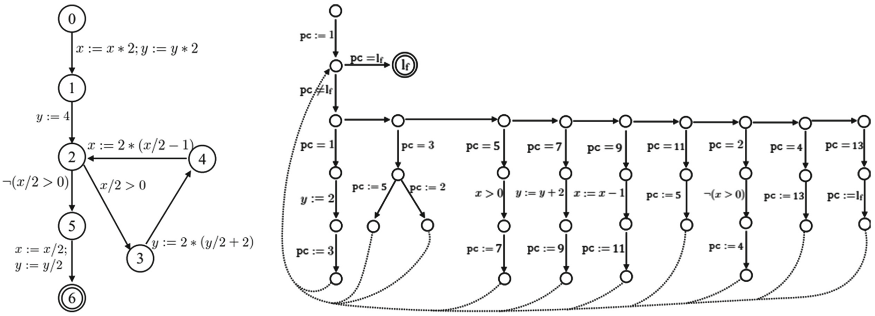


Fig. 8. Obfuscated CFGs.

Theorem 4. *Let* $\mathtt{int}_{Par} \stackrel{\text{def}}{=} \langle \mathtt{D}_{Par} \diamond \mathtt{cfgEx}, [\![\cdot]\!]_{\mathfrak{C}} \rangle$ *be the distorted interpreter. It is a Par-distorting interpreter, meaning that* $\forall \mathsf{P} \in \mathfrak{L}$. *spec*$(\mathtt{int}_{Par}, \mathsf{P})$ *is an obfuscation of* P *w.r.t* $\mathcal{A}_{Par}$.

Control Obfuscation: CFG Flattening. The last example consists in obfuscating the CFG by flattening its structure [7,36]. For obfuscating programs by flattening the CFG it is sufficient to make the program counter (*pc*) dynamic, i.e., a variable of the program manipulated during execution. Indeed, in this way the CFG observation (and therefore any CFG property) becomes imprecise [20]. Hence the idea is precisely to provide the transformers distorting the interpreter by handling the program counter while executing the program.

Then, let us consider the new variable *pc*, we define the distorted parser by inserting each statement in a branch of a non-deterministic choice, whose guard is the value of *pc*, value that is created and updated during execution. Also this distorter $\mathtt{D}_{Flat}$ (Fig. 9) cannot be simply a pushdown automaton for two issues to face. First, we have to *count* the number of deterministic choices we insert, in order to know how many brackets $\rangle_+$ we have to insert at the end (q_4). The stack *cl* is used precisely with this purpose. The other issue to face, is the fact that when we read ${}_*\langle$ or ${}_+\langle$ we cannot know how many statement we will have respectively in the body or in the first branch, hence we cannot predict the value for *pc* that we can use for skipping the loop or for the other branch. For this reason we use two stacks *p* (principal) and *s* (secondary) in order to keep two disjoint chains of values for *pc* (even and odd values). Finally we have another stack *c* for keeping trace of the *pc* of the first statement of a loop and of the final *pc* of a branch. Moreover, we consider a special value $\mathtt{l_f}$ as final *pc*.

On the right of Fig. 8 the obfuscation of the program whose CFG is depicted in Fig. 2.

Formally, let us characterize the semantic abstraction made incomplete by flattening the program. Here we can simplify the previous characterization [20] since it is sufficient to find an abstraction made incomplete by a dynamic *pc*

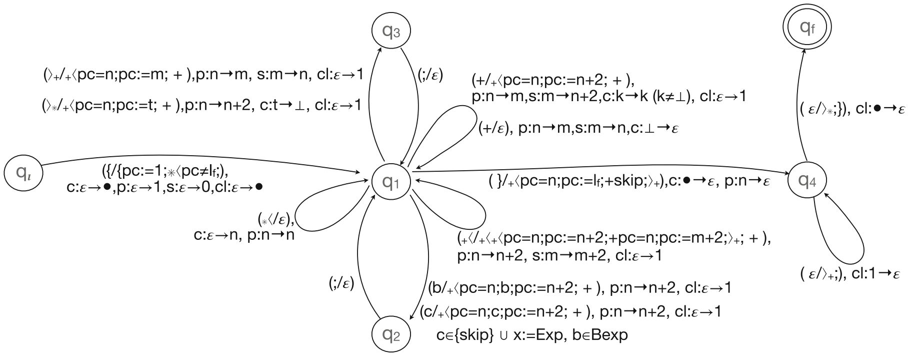

q3
qf
(⟩+/+⟨pc=n;pc:=m; +),p:n→m, s:m→n, cl:ε→1
(⟩*/+⟨pc=n;pc:=t; +),p:n→n+2, c:t→⊥, cl:ε→1
(;/ε)
(+/+⟨pc=n;pc:=n+2; +),
p:n→m,s:m→n+2,c:k→k (k≠⊥), cl:ε→1
(+/ε), p:n→m,s:m→n,c:⊥→ε
(ε/⟩*;}), cl:●→ε
qι
({/{pc:=1;*⟨pc≠lf;),
c:ε→●,p:ε→1,s:ε→0,cl:ε→●
q1
(}/+⟨pc=n;pc:=lf;+skip;⟩+),c:●→ε, p:n→ε
q4
(*⟨/ε),
c:ε→n, p:n→n
(+⟨/+⟨+⟨pc=n;pc:=n+2;+pc=n;pc:=m+2;⟩+; +),
p:n→n+2, s:m→m+2, cl:ε→1
(ε/⟩+;), cl:1→ε
(;/ε)
(b/+⟨pc=n;b;pc:=n+2; +), p:n→n+2, cl:ε→1
(c/+⟨pc=n;c;pc:=n+2; +), p:n→n+2, cl:ε→1
q2
c∈{skip} ∪ x:=Exp, b∈Bexp

Fig. 9. Flattening obfuscation interpreter distorter $\mathtt{D}_{Flat}$.

(we are not looking for an abstraction incomplete *iff* the *pc* is dynamic), and it should be clear that such an abstraction is precisely the CFG: Let us define

$$\mathcal{A}^{Flat} \overset{\text{def}}{=} \lambda [\![\mathsf{P}]\!]_{\mathcal{C}}.\ [\![\mathtt{Cfg}(\mathsf{P})]\!]_{\mathcal{C}}$$

where it is well known that program collecting semantics is an abstraction of its small-step semantics and

$$[\![\mathtt{Cfg}(\mathsf{P})]\!]_{\mathcal{C}} \overset{\text{def}}{=} \left\{ \sigma \in \mathbb{S}^* \,\middle|\, \begin{array}{l} \forall i \in \mathbb{N}.\ f^{\mathtt{cfgEx[P]}}_{\mathtt{CollRsp}}(q_i, \sigma_i) \in \{\langle q_{i+1}, \sigma_{i+1}\rangle, \sigma_{i+1}\} \\ q_0 = q_\iota,\ \sigma_0 = [q_\iota \mapsto \mathbb{M}] \end{array} \right\}$$

In this way, it should be clear that $\mathtt{Cfg}(\mathsf{P})$ is an abstraction of P, since the set of possible executions contains the set of executions[14].

The following theorem tells us that in order to lose any property of the CFG structure we need to handle the program counter (*pc*) deciding the next statement to execute in the program. Intuitively in this way the semantic abstraction, by abstracting the guards involving the *pc*, loses the CFG structure, and therefore any of its properties.

Theorem 5. *Let* $\mathtt{int}_{Flat} \overset{\text{def}}{=} \langle \mathtt{D}_{Flat} \diamond \mathtt{cfgEx}, [\![\cdot]\!]_{\mathcal{C}}\rangle$ *be the distorted interpreter. It is a* $\mathcal{F}$*-distorting interpreter, meaning that* $\forall \mathsf{P} \in \mathfrak{L}$. *spec*$(\mathtt{int}_{Flat}, \mathsf{P})$ *is an obfuscation of* P *w.r.t* $\mathcal{A}^{Flat}$.

7 Conclusions

Our paper shows that code transformations for anti reverse engineering, such as code obfuscating transformations, can be designed systematically by specializing distorted interpreters. The idea proposed is to design an obfuscation starting from the static property π to conceal and determining syntactic elements such

[14] We can also let the computations start from a subset of initial memories.

that the static analysis of π is imprecise/incomplete. Then we can build a distorter D inductively embedding these elements in single statements of programs without changing their semantics. Finally the distorted interpreter we obtain so far becomes an obfuscator of π when specialized w.r.t. the program to obfuscate, since for the resulting program the static analysis of π is forced to be incomplete. The result is a systematic method for building obfuscating compilers as straight applications of the well known Futamura's projections, therefore going beyond the very first approach [20].

On the Limits of the Approach. The main limit of the proposed approach consists in the fact that it is built on a static model of attacker, namely aiming at defeating reverse engineering techniques based on static program analyses. In particular, on the one hand we believe that it is possible to capture any obfuscation technique where the information to obfuscate is a (static) program property that can be characterized as program/code abstraction, such as slicing obfuscation and opaque predicate obfuscation, on the other hand there probably exist dynamic obfuscation techniques that may not be modeled in the framework as it is, such as self-modifying code and virtualization. Indeed, what we propose is an interpreter distortion based on the distortion of the parser, which is a distortion limited to *obfuscating properties strictly depending on how the code is written*, namely for deceiving static analyses. Nevertheless, the separation of the static and the dynamic phases of the interpreter suggests us that we could try to capture other obfuscation techniques by distorting the semantic interpretation phase, where it is plausible to think of making possible to obfuscate dynamic analyses, but this clearly needs further work.

Another limit of this work is that it is not yet evaluated by means of an implementation. Anyway we believe that this will not represent a problem at least as far as the computational impact is concerned, since once we have built the interpreter depending on the property to obfuscate, then the complexity of the specialization is linear on the length of the program (transformation is performed while parsing the code). While, as far as the computational overhead of the obfuscated program is concerned, it does not depend on the proposed approach but on the way the code is transformed, and therefore on the specific chosen transformation.

Future Works. The notion of Jones optimality can be seen in a wider perspective, not just as a method for removing the overhead in time complexity, but as an universal paradigm to determine when a specializer applied to an interpreter and a program reestablish the initial conditions of the program relatively to some measure [6]. Hence, the existence of Jones optimal specializers is a key aspect in modern PL research (e.g., see the Brown and Palsberg striking result [3]). We believe that widening the range of applicability of the notion of Jones optimality to different (complexity) measures may provide the perfect theoretical framework to understand how the intensional nature of code affects the way we analyze it. Our paper provides a very first application to the case of the precision of an abstract interpreter.

References

1. Arceri, V., Mastroeni, I.: Analyzing dynamic code: a sound abstract interpreter for evil eval. ACM Trans. Priv. Secur. **24**(2), 10:1–10:38 (2021)
2. Barak, B., et al.: On the (im)possibility of obfuscating programs. J. ACM **59**(2), 6 (2012)
3. Brown, M., Palsberg, J.: Jones-optimal partial evaluation by specialization-safe normalization. Proc. ACM Program. Lang. **2**(POPL), 14:1–14:28 (2018). https://doi.org/10.1145/3158102
4. Bruni, R., Giacobazzi, R., Gori, R., Garcia-Contreras, I., Pavlovic, D.: Abstract extensionality: on the properties of incomplete abstract interpretations. Proc. ACM Program. Lang. **4**(POPL), 28:1–28:28 (2020). https://doi.org/10.1145/3371096
5. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: Symposium on Logic in Computer Science, LICS, pp. 1–13. IEEE (2021)
6. Campion, M., Dalla Preda, M., Giacobazzi, R.: Partial (in)completeness in abstract interpretation: limiting the imprecision in program analysis. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498721
7. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.C.: A white-box DES implementation for DRM applications. In: Feigenbaum, J. (ed.) DRM 2002. LNCS, vol. 2696, pp. 1–15. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-44993-5_1
8. Collberg, C., Herzberg, A., Gu, Y., Giacobazzi, R., Wang, F., Davidson, J.: Toward digital asset protection. IEEE Intell. Syst. **26**(06), 8–13 (2011). https://doi.org/10.1109/MIS.2011.106
9. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional, Boston (2009)
10. Collberg, C., Thomborson, C.: Watermarking, tamper-proofing, and obduscation-tools for software protection. IEEE Trans. Software Eng. 735–746 (2002)
11. Consel, C., Lawall, J., Meur, A.F.L.: A tour of Tempo: a program specializer for the C language. Sci. Comput. Program. **52**(17(1)), 47–92 (2004)
12. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. **277**(1–2), 47–103 (2002)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977), pp. 238–252. ACM Press (1977)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the 6th ACM Symposium on Principles of Programming Languages (POPL 1979), pp. 269–282. ACM Press (1979)
15. Cousot, P., Giacobazzi, R., Ranzato, F.: A^{2}i: abstract^2 interpretation. Proc. ACM Program. Lang. **3**(POPL), 42:1–42:31 (2019)
16. Dalla Preda, M., Mastroeni, I.: Characterizing a property-driven obfuscation strategy. J. Comput. Secur. **26**(1), 31–69 (2018)
17. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: POPL, pp. 541–554 (2014)
18. Drape, S.: Obfuscation of abstract data-types. Ph.D. thesis, University of Oxford (2004)

19. Giacobazzi, R.: Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In: Proceedings of the 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM 2008), pp. 7–20. IEEE Press (2008)
20. Giacobazzi, R., Jones, N.D., Mastroeni, I.: Obfuscation by partial evaluation of distorted interpreters. In: Kiselyov, O., Thompson, S. (eds.) Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2012), pp. 63–72. ACM Press (2012)
21. Giacobazzi, R., Mastroeni, I., Dalla Preda, M.: Maximal incompleteness as obfuscation potency. Formal Asp. Comput. **29**(1), 3–31 (2017)
22. Giacobazzi, R., Ranzato, F., Scozzari., F.: Making abstract interpretation complete. J. ACM **47**(2), 361–416 (2000)
23. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015, pp. 261–273. ACM (2015)
24. Giacobazzi, R., Pavlovic, D., Terauchi, T.: Intensional and extensional aspects of computation: From computability and complexity to program analysis and security (NII Shonan Meeting 2018-1). In: NII Shonan Meeting Report 2018 (2018)
25. Glück, R.: Jones optimality, binding-time improvements, and the strength of program specializers. In: Asai, K., Chin, W. (eds.) Proceedings of the ACM SIGPLAN ASIA-PEPM 2002, Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Aizu, Japan, 12–14 September 2002, pp. 9–19. ACM (2002)
26. Jones, N.D.: Transformation by interpreter specialisation. Sci. Comput. Programm. **52**(17(1)), 307–339 (2004)
27. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River (1993)
28. Jørgensen, J.: Similix: a self-applicable partial evaluator for scheme. In: Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 83–107. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-47018-2_3
29. Leuschel, M.: Advanced logic program specialisation. In: Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 271–292. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-47018-2_11
30. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. (POPL) **4**(10) (2020)
31. Rogers, H.: Theory of Recursive Functions and Effective Computability. The MIT Press, Cambridge (1992)
32. Romanenko, S.: Unmix, a Specializer for a Subset of Scheme. Keldysh Institute, Moscow (1993–2009). http://code.google.com/p/unmix/
33. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design - Analysis and Transformation. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-17548-0
34. Thiemann, P.: Aspects of the PGG system: specialization for standard scheme. In: Hatcliff, J., Mogensen, T.Æ., Thiemann, P. (eds.) DIKU 1998. LNCS, vol. 1706, pp. 412–432. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-47018-2_17
35. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic finite state transducers: algorithms and applications. In: POPL, pp. 137–150 (2012)
36. Wang, C., Davidson, J., Hill, J., Knight, J.: Protection of software-based survivability mechanisms. In: IEEE International Conference of Dependable Systems and Networks, Goteborg, pp. 193–202 (2001)
37. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)