Annotated multisemantics to prove Non-Interference analyses
Gurvan Cabon, Alan Schmitt

To cite this version:

HAL Id: hal-01656404
https://hal.archives-ouvertes.fr/hal-01656404
Submitted on 20 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Annotated multisemantics to prove Non-Interference analyses

Gurvan Cabon
Inria
Rennes, France

Alan Schmitt
Inria
Rennes, France

ABSTRACT
The way information flows into programs can be difficult to track. As non-interference is a hyperproperty relating the results of several executions of a program, showing the correctness of an analysis is quite complex. We present a framework to simplify the certification of the correction proof of such analyses. The key is capturing the non-interference property through an annotated semantics based on the execution of the program and not simply its result. The approach is illustrated using a small While language.

CCS CONCEPTS
• Security and privacy → Formal methods and theory of security;

KEYWORDS
multisemantics, non-interference, pretty-big-step, annotation

1 INTRODUCTION
Non-interference can be defined as a program property that give guarantees on the independence of specific (public) outputs of a program from specific (secret) inputs. Non-interference is a hyperproperty [9]: it does not depend on one particular execution of the program (unlike illegal memory access for example), but on the results of several executions.

To develop a certified system verifying information flows, such as non-interference, we propose to only rely on the execution of the program, and thus investigate such properties using directly the derivation tree of an execution.

Considering a single execution is clearly not sufficient to determine if a program has the non-interference property. Surprisingly, studying every execution independently is also not sufficient. This is why we propose a formal approach that builds, from any semantics respecting a certain structure, a multisemantics that allows to reason on several executions simultaneously. Adding annotations to this multisemantics lets us capture the dependencies between inputs and outputs of a program.

We show that our approach is correct, i.e., annotations correctly capture non-interference. This allows analyses (systems detecting information leak giving non-interferent guarantees only when the tested program is actually non-interferent) to be proven correct as the dependencies are a simple property of the multisemantics defined by induction.

2 NON-INTERFERENCE
Suppose we have a programming language in which variables can be private or public, and where the programs can take variables as parameters. We say a program is non-interferent if, for any pair of execution that differs only on the private parameters, the values of the public variables are the same. In other words, changing the value of the private variables does not influence the public variables. Or in yet other words, the public variables do not depend on the private variables: there is no leak of private information.

Definition 2.1 (Termination-Insensitive Non-interference).
A program is Non-interferent if, for any pair of terminating executions starting with different values in the private variables, the executions end with the same value in the public variables.

In this work, we only consider finite program executions. We now illustrate through examples of increasing complexity where leaks of private information may happen and how one may detect them. As a simple first example, consider the naive program in Figure 1, where public is a public parameter and secret is a private variable. It is clearly interferent (or not non-interferent): changing the value of secret changes the value of public. This is a direct flow of information because the value of secret is directly assigned into public.

public := secret

Figure 1: Example of naive interference

Unfortunately, interference is not simply the transitive closure of direct flows. It may also come from the context in which a particular
instruction is executed. For example, Figure 2 shows a program with an indirect flow. The value of secret is not directly stored into public but the condition in the if statement ensures that in each case secret receives the value of public. One may thus detect interference by taking into account the context in which an assignment takes place. Any single execution of the program of Figure 2 would then witness the interference.

```python
if secret
    then public := true
else public := false
```

**Figure 2: Example of indirect flow**

Another source of interference is the fact that not executing a part of the code can provide information. This is often called masking. For example, Figure 3 shows such a program. In the case where secret is false, the variable public is not modified, so this execution does not witness the interference, even when taking the context into account. The other execution, where secret is true, does witness the interference. Hence, a further refinement to detect interference would be to consider all possible executions of a program.

```python
public := false
if secret
    then public := true
else skip
```

**Figure 3: Example of indirect flow with a mask**

Unfortunately, this is not sufficient. In the example shown in Figure 4, we can see that there exists no single execution where the flow can be inferred. In the left execution, public depends on y, which is not modified by the execution. In the right execution, public still depends on y, which itself depends by indirect flow on x, which is not modified by the execution. Hence in both cases there seems to be no dependency on secret. Yet, we have public = secret at the end of both execution, so the secret is leaked. Looking at every execution independently is not enough.

To recover the inference of information flow as a property of an execution, we propose a different semantics where multiple executions are considered in lock-step, so that one may combine the information gathered by several executions. In the case of Figure 4, we can see that x depends on secret in the first execution at the end of the first if. Hence, in the second execution, x must also depend on secret, as the fact that not modifying it is an information flow. We can similarly deduce that y depends on x in both executions, hence public transitively depends on secret.

In some sense, we propose to internalize an approximation of the non-interference hyperproperty in a property of a refined semantics. Our approach gives the ability to reason inductively on the refined semantics and construct formal proofs of correctness of analyses.

3 PRETTY-BIG-STEP

As we aim to provide a generic framework independent of a specific programming language, we need a precise and simple way to describe its semantics. The Pretty-Big-Step semantics [8] is not only concise, it has been shown to scale to complex programming languages while still being amenable to formalization with a proof assistant [7]. We slightly modify the definition of Pretty-Big-Step to make it more uniform and to simplify the definition of non-interference.

3.1 Canonical structure

**Memory model.** We propose to model non-interference by making explicit the inputs of a program and its outputs. We do not consider interactive programs, so each input is a constant single value, for instance an argument of the program. Outputs, however, consist of lists of values, as we allow a program to send several values to a given output.

Formally, we consider given a set of values Val and a set of variables Var. We define the memory as a triplet \((E_I, E_x, E_o)\), where \(E_I \in Env_I\) represents the inputs of a program as a read-only mapping from each input to a value, \(E_x \in Env_x\) represents run-time environment as a read-write mapping from each variable to a value, and \(E_o \in Env_o\) represents the outputs of a program, as a write-only mapping of each output to a list of values, accumulated in the output. To simplify, we consider inputs and outputs to be indexed by an integer.

\[
\begin{align*}
\text{Inputs} & := \mathbb{N} \\
\text{Outputs} & := \mathbb{N} \\
\text{Env}_I & := \text{Inputs} \rightarrow \text{Val} \\
\text{Env}_x & := \text{Var} \rightarrow \text{Val} \\
\text{Env}_o & := \text{Outputs} \rightarrow \text{Val list} \\
\text{Mem} & := \text{Env}_I \times \text{Env}_x \times \text{Env}_o
\end{align*}
\]
**Semantics.** The Pretty-Big-Step semantics is a constrained Big-Step semantics where each rule may only have 0, 1, or 2 inductive premises. In addition, one only needs to know the state and term under evaluation to decide which rule applies. To illustrate the Pretty-Big-Step approach, let us consider the evaluation of a conditional. It may look like this is Big-Step format.

\[
\frac{M, e \rightarrow (M', v)}{M, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow M''}
\]

Although this rule only has two inductive premises, one has to partially execute it to know if it is applicable (in this case if \(e\) evaluates to \(true\)). In Pretty-Big-Step, one first evaluates \(e\), then passes control to another rule to decide which branch to evaluate. Additional constructs are needed to describe these intermediate steps, they are called extended terms, often written with a \(1\) or \(2\) subscript, and they need previously computed values. Here are the rules for evaluating a conditional in Pretty-Big-Step.

\[
\frac{M, e \rightarrow (M', v)}{M, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightarrow M''}
\]

\[
\frac{M, s_1 \rightarrow M'}{(M, \text{true}), \text{if } s_1 \text{ s}_2 \rightarrow M''}
\]

Formally, rules are in three groups shown in Figure 5: (i) axioms, the rules with no inductive premise; (ii) rules 1, the rules with one inductive premise; (iii) rules 2, the rules with two inductive premises.

Rules may either return a memory and a value, or just a memory. Conversely, in Pretty-Big-Step, rules may take as input a memory and zero, one, or several values. To account for this in a uniform way, we define a state \(\sigma\) as a pair of a memory and a list of values, called an extra. We write \(\text{extra}(\sigma)\) to refer to the list of values in a state \(\sigma\). The result of evaluating an expression is a state whose extra is a singleton list containing the resulting value. To simplify notations, we omit the extra when it is an empty list.

\begin{align*}
\text{Extra} & := \text{List(Val)} \\
\text{State} & := \text{Mem} \times \text{Extra}
\end{align*}

A rule is entirely defined by the following components.

- **Axioms**
  - \(t : \text{term}\), the term on which the axiom can be applied;
  - \(\text{ax} : \text{State} \rightarrow \text{State option}\), a function that gives the resulting state given the initial state.

- **Rule 1**
  - \(t : \text{term}\), the term on which the rule 1 can be applied;
  - \(\text{up} : \text{State} \rightarrow \text{State option}\), a function returning the new state in which \(t_1\) will be evaluated;
  - \(t_1 : \text{term}\), a term to evaluate in order to continue the derivation.

- **Rule 2**
  - \(t : \text{term}\), the term on which the rule 2 can be applied;
  - \(\text{up} : \text{State} \rightarrow \text{State option}\), a function returning the state in which the term \(t_1\) has to be applied;
  - \(t_1 : \text{term}\), a function giving the state in which \(t_1\) had to be derived depending on the initial state and the result of the derivation of \(t_1\);
  - \(t_1, t_2 : \text{term}\), the terms to derive in order to get the result for \(t\);
  - \(\text{prod}_\text{extra}\), a boolean value indicating if the evaluation of \(t_1\) produces an extra.

The functions \(\text{ax}, \text{up}, \text{and next}\) are functions returning a State option because these functions have no image for some states. For example, the rule \(\text{IfTrue}\) above is defined only when the state has a single extra that is the boolean value \(true\). The \(\text{prod}_\text{extra}\) boolean is used to distinguish rules that produce an intermediate state with a non-empty extra to those who produce one with an empty extra. It is only used in Section 5 when annotating rules.

\[
\text{Ax} : \sigma, t \rightarrow \text{ax}(\sigma) \\
\text{R}_1 : \sigma, t \rightarrow \sigma' \\
\text{R}_2 : \up(\sigma), t_1 \rightarrow \sigma'_1 \quad \text{next}(\sigma, \sigma'_1), t_2 \rightarrow \sigma'
\]

**Figure 5:** Types of rule for a Pretty-Big-Step semantics

For clarity reasons, Figure 5 assumes \(\text{ax}(\sigma), \up(\sigma)\) and \(\text{next}(\sigma, \sigma'_1)\) return actual states and not an optional states. Rules are not defined when the results are None.

The intuition behind the rules Pretty-Big-Step is the following.

- If the evaluation is immediate, we can directly give the result (e.g., the evaluation of a skip statement or a constant). This behavior corresponds to an axiom.

- If the evaluation needs to branch depending on a previously computed value, stored as an extra, then a rule 1 is used. This is used for instance after evaluating the condition in a conditional statement.

- If the evaluation needs to first inductively compute an intermediate result, then a rule 2 is used. The intermediate result is used to compute the next state with which the evaluation continues.

We thus impose the following additional requirements. For rules 1 and rules 2, if \(\up\) is defined, then it must not change nor inspect the memory, i.e., it can only change the extra part of the state, and this change is a function of the previous extra: \(\up(M, e) = \text{Some}(M', e') \implies M' = M \land e' = f(e)\). For rules 2, if \(\text{next}\) is defined, then the new memory is the memory of the second argument, and the new extra only depends on the extra of the arguments: \(\text{next}((M_1, e_1), (M_2, e_2)) = \text{Some}(M, e) \implies M = M_2 \land e = g(e_1, e_2)\). Finally, given a term and an extra, at most one rule applies.

### 3.2 WHILE language

To illustrate our approach, we introduce a small WHILE language.

In this language, we distinguish two kinds of terms: expressions and statements. We first give the syntax of the language and then its semantics in Pretty-Big-Step form.

**Syntax.** An expression is either a constant value, a variable, an input, or the binary operation between two expressions. A statement is either a no-op operation skip, a sequence of two statements, a
We refer to such a µ.

We use some usual notations.

Semantics. To simplify the reading of the rules and the examples, we use some usual notations.

c for Const c
x for Var x
e₁ op e₂ for Op e₁ e₂
s₁; s₂ for Seq s₁ s₂
x := e for Assign x e
x := f for Assign 1 x
if e then s₁ else s₂ for If e s₁ s₂
If t₁ s₁ s₂ for If1 t₁ s₁ s₂
while e s for While e s
while e do s for While1 e s
while e do s for While2 e s

f[x ↦ v] denotes the function y ↦ \{ v if x = y \\ f(y) otherwise \}.

As an example of the Pretty-Big-Step semantics, consider the conditional, a while loop, an assignment of an expression into a variable, or an assignment of an expression into an output.

expression e ::= Const n | Var x | Input n | Op e e

statement s ::= Skip | Seq s s | If e s s | While e s | Assign x e | Output n e

We add to the expressions and statements the extended terms for

\[ \mu \]

... | Op1 e | Op2

statement s ::= ... | Seq1 s | If1 s s | While1 e s | While2 e s | Assign1 x | Output1 n

Semantics. To simplify the reading of the rules and the examples, we use some usual notations.

c for Const c
x for Var x
e₁ op e₂ for Op e₁ e₂
s₁; s₂ for Seq s₁ s₂
x := e for Assign x e
x := f for Assign 1 x
if e then s₁ else s₂ for If e s₁ s₂
If t₁ s₁ s₂ for If1 t₁ s₁ s₂
while e s for While e s
while e do s for While1 e s
while e do s for While2 e s

f[x ↦ v] denotes the function y ↦ \{ v if x = y \\ f(y) otherwise \}.

As an example of the Pretty-Big-Step semantics, consider the conditional, a while loop, an assignment of an expression into a variable, or an assignment of an expression into an output.

expression e ::= Const n | Var x | Input n | Op e e

statement s ::= Skip | Seq s s | If e s s | While e s | Assign x e | Output n e

We add to the expressions and statements the extended terms for

\[ \mu \]

... | Op1 e | Op2

statement s ::= ... | Seq1 s | If1 s s | While1 e s | While2 e s | Assign1 x | Output1 n

Semantics. To simplify the reading of the rules and the examples, we use some usual notations.

c for Const c
x for Var x
e₁ op e₂ for Op e₁ e₂
s₁; s₂ for Seq s₁ s₂
x := e for Assign x e
x := f for Assign 1 x
if e then s₁ else s₂ for If e s₁ s₂
If t₁ s₁ s₂ for If1 t₁ s₁ s₂
while e s for While e s
while e do s for While1 e s
while e do s for While2 e s

f[x ↦ v] denotes the function y ↦ \{ v if x = y \\ f(y) otherwise \}.

As an example of the Pretty-Big-Step semantics, consider the conditional, a while loop, an assignment of an expression into a variable, or an assignment of an expression into an output.

expression e ::= Const n | Var x | Input n | Op e e

statement s ::= Skip | Seq s s | If e s s | While e s | Assign x e | Output n e

We add to the expressions and statements the extended terms for

\[ \mu \]

... | Op1 e | Op2

statement s ::= ... | Seq1 s | If1 s s | While1 e s | While2 e s | Assign1 x | Output1 n

Semantics. To simplify the reading of the rules and the examples, we use some usual notations.

c for Const c
x for Var x
e₁ op e₂ for Op e₁ e₂
s₁; s₂ for Seq s₁ s₂
x := e for Assign x e
x := f for Assign 1 x
if e then s₁ else s₂ for If e s₁ s₂
If t₁ s₁ s₂ for If1 t₁ s₁ s₂
while e s for While e s
while e do s for While1 e s
while e do s for While2 e s

f[x ↦ v] denotes the function y ↦ \{ v if x = y \\ f(y) otherwise \}.

As an example of the Pretty-Big-Step semantics, consider the conditional, a while loop, an assignment of an expression into a variable, or an assignment of an expression into an output.

expression e ::= Const n | Var x | Input n | Op e e

statement s ::= Skip | Seq s s | If e s s | While e s | Assign x e | Output n e

We add to the expressions and statements the extended terms for

\[ \mu \]

... | Op1 e | Op2

statement s ::= ... | Seq1 s | If1 s s | While1 e s | While2 e s | Assign1 x | Output1 n

4 MULTISEMANTICS

The first step of our approach is to derive a new semantics where

The same rule at the same point in the derivation are shared.

several derivations are considered at once. We do not simply want

to derive a rule in Pretty-Big-Step style. There are three cases as there are

three kinds of Pretty-Big-Step rules.

In order to derive an axiom, the multistate should be consistent

with the ð function for every pair, that is for every pair

σ₁, σ₂ of a relation obtained by derivation of t₁. To
derive a rule 2, for every pair (σ, σ′) ∈ µ, their should exists three

states σ₁, σ′₁, σ₂ such that:

• up(σ) is of the form Some σ₁
• (σ₁, σ′₁) is a pair of a multistate obtained by derivation of t₁
• next(σ, σ′₁) is of the form Some σ₂
• (σ₂, σ′) is a pair of a multistate obtained by derivation of t₂

\[ \mu = ax_{Some}(f(\mu)) μ \neq 0 \]

\[ t \downarrow \mu \]

\[ t_1 \downarrow \mu_1 \]

\[ t_2 \downarrow \mu_2 \]

\[ \mu_2 = \mu'_n \circ next_{Some}(\mu'_n) \circ \mu_2 \]

\[ t \downarrow \mu \]
Because we need \( \sigma \) to determine \( \text{next}(\sigma, \sigma_1') \), we use the \( \rightarrow \) operator to remember \( \sigma \).

These rules are not sufficient in the general case as they force every derivation to have the same structure. For example, when trying to derive an \( if \) statement in the multisemantics, all of the derivations have to go in the same branch. The multiderivation for a conditional has the following root.

\[
\begin{align*}
&b \upharpoonright \mu_1 \\
&\text{If } s_1 \ s_2 \downarrow \mu_2 \\
&\mu_n = \upordinate{\text{first}(\mu)} \circ \mu_1 \\
&\mu = \mu_n \circ \text{next}(\text{snd}(\mu_n)) \circ \mu_2 \\
&\text{if } b \text{ then } s_1 \text{ else } s_2 \downarrow \mu
\end{align*}
\]

To derive \( \text{If } s_1 \ s_2 \downarrow \mu \) there are two options. Either

\[
\begin{align*}
&\text{IfTrue} \\
&s_1 \downarrow \mu_1 \\
&\mu = \upordinate{\text{first}(\mu)} \circ \mu_1 \\
&\text{IfTrue} \\
&s_2 \downarrow \mu_2 \\
&\mu = \upordinate{\text{first}(\mu)} \circ \mu_2
\end{align*}
\]

where \( \upordinate{\text{first}(\mu)} = \{(M, \text{true}), M\} \) if \( (M, \text{true}) \in \text{first}(\mu) \) and \( \text{first}(\mu) \) only contains states of the form \( (M, \text{true}) \), or

\[
\begin{align*}
&\text{IfFalse} \\
&s_1 \downarrow \mu_1 \\
&\mu = \upordinate{\text{first}(\mu)} \circ \mu_1 \\
&\text{IfFalse} \\
&s_2 \downarrow \mu_2 \\
&\mu = \upordinate{\text{first}(\mu)} \circ \mu_2
\end{align*}
\]

where \( \upordinate{\text{first}(\mu)} = \{(M, \text{false}), M\} \) if \( (M, \text{false}) \in \mu \) and \( \text{first}(\mu) \) only contains states of the form \( (M, \text{false}) \).

As those options are incompatible, it is impossible to have a multiderivation for a conditional where the guard is evaluated differently for some states. To fix this, we add a Merge rule. This rule simply states that if it is possible to derive a term with two multistates, then it is also possible to derive it from the union of them. In the case of an \( if \) statement \( if \), one may thus use two subderivations, one for each status of the guard, and merge them together.

\[
\begin{align*}
&\text{Merge} \\
&t \upharpoonright \mu_1 \\
&t \upharpoonright \mu_2 \\
&\text{Merge} \\
&t \upharpoonright \mu_1 \cup \mu_2
\end{align*}
\]

We do not restrict the use of the Merge rule. In practice, we only use it when we need to apply different rules to a multistate.

### 4.2 Expected properties

We now prove properties that show that multiderivations correspond to multiple derivations. First, if \( t \downarrow \mu \) is derivable, then for every pair \( (\sigma, \sigma') \in \mu, \sigma, t \rightarrow \sigma' \) is derivable. A proof by induction on the multiderivation is straightforward.

**Lemma 4.1.** \( \forall \mu. \ t \downarrow \mu \implies \forall (\sigma, \sigma') \in \mu. \sigma, t \rightarrow \sigma' \)

The converse implication is not true, however. Figure 8 shows an example of a program allowing Pretty-Big-Step derivations of
arbitrary size. For every $k \in \mathbb{N}$, a derivation starting with the value $k$ in the first input needs to unroll $k$ times the while loop. Each of these derivations are finite but considering all of them together would require an infinite multiderivation.

\[
\begin{align*}
n &:= \text{In } 1 \\
i &:= 0 \\
\text{While } (i < n) \text{ do} \\
i &:= i + 1
\end{align*}
\]

Figure 8: Counter example to the reciprocal of lemma 4.1

Nonetheless, when taking a finite number of Pretty-Big-Step derivations, we are able to derive them all together in the multi-semantics. Using the fact that a finite set can be described as the union of singletons (one for each element of the set), we can prove this with the two lemmas 4.2 and 4.3. The first one states that if a term is derivable in Pretty-Big-Step then it is derivable in the multisemansics with the corresponding set of iterations. The second lemma states that if a term is derivable with two multistates then it is derivable with the union of them. Finite multistates are sufficient for our purpose since finding interference only requires two derivations (or equivalently: proving non-interference only requires to inspect every pair of derivations).

\begin{align*}
\text{LEMMA 4.2. } \forall \sigma, \sigma'. \sigma \rightarrow \sigma' &\implies t \Downarrow (\sigma, \sigma') \\
\text{LEMMA 4.3. } \forall \mu_1, \mu_2. t \Downarrow \mu_1 &\implies t \Downarrow \mu_2 \implies t \Downarrow \mu_1 \cup \mu_2
\end{align*}

The first lemma is proved by induction on the Pretty-Big-Step derivation and the second one is a direct use of the Merge rule. We have formally proved these three lemmas in Coq.

5 ANNOTATIONS

We now present how multiderivations may be annotated to track information flows.

5.1 Construction of the annotations

Our annotations track the inputs on which every variable and output depends in a dependency environment of type $\text{Dep}$, typically written $D$. Additionally, we track the context dependency $\text{CD}$ of the current computation. It has type $\text{CtxtDep}$, a set of inputs, and it represents the dependency of the context in which the current expression or statement is evaluated. The context dependency is used to track indirect flows, and is similar to program counter levels, although more precise.

\[\text{Dep} := (\text{Var} \cup \text{Outputs}) \rightarrow \text{Inputs set}\]
\[\text{CtxtDep} := \text{Inputs set}\]

An annotated derivation is written as follows.

\[CD, D, t \Downarrow \mu, D', VD'\]

$CD \in \text{CtxtDep}$ and $D \in \text{Dep}$ are the context dependency and the dependency environment before the execution. $D' \in \text{Dep}$ is the dependency environment after the execution of the term. $VD' \in \text{CtxtDep}$ is the set of inputs the computed value, i.e., the extra, depends on.

We suppose we are given, for each axiom, the inputs, variables, and outputs used by the rule. Formally, each axiom comes with four sets:

- $\text{InputRead} \subset \text{Inputs}$, the set of inputs the axiom may read;
- $\text{VarRead} \subset \text{Var}$, the set of variables the axiom may read;
- $\text{VarWrite} \subset \text{Var}$, the set of variables the axiom may write;
- $\text{OutputWrite} \subset \text{Outputs}$, the set of outputs the axiom may write.

These sets respect the following properties.

1. If two states have identical extras, and their memories are equal on the inputs and variables that can be read by the axiom, then for every variable $x \in \text{VarWrite}$, the value stored in $x$ after the axiom is the same in both memories.
2. The value in variables not in $\text{VarWrite}$ are not modified by the axiom.
3. If two states have identical extras, and their memories are equal on the inputs and variables that can be read by the axiom, then for every output $o \in \text{OutWrite}$, the value added to $o$ after the axiom is the same in both memories.
4. The value in outputs not in $\text{OutWrite}$ are not modified by the axiom.

More formally:

\begin{enumerate}
\item $\forall \sigma \sigma'. \sigma \rightarrow \sigma' \implies (\sigma, \sigma') \in \text{extra} \Rightarrow t \Downarrow ((\sigma, \sigma'))$
\item $\forall \sigma_1, \sigma_2. (\text{extra} \sigma_1) = (\text{extra} \sigma_2) \Rightarrow (\forall i \in \text{InputRead}. \sigma_1(i) = \sigma_2(i))$
\item $\forall \forall y \in \text{VarRead}. \sigma_1(y) = \sigma_2(y) \Rightarrow (\forall x \in \text{VarWrite}. \text{ax} \sigma_1(x) = \text{ax} \sigma_2(x))$
\item $\forall \sigma_1, \sigma_2. (\text{extra} \sigma_1) = (\text{extra} \sigma_2) \Rightarrow (\forall i \in \text{InputRead}. \sigma_1(i) = \sigma_2(i))$
\item $\forall y \in \text{VarRead}. \sigma_1(y) = \sigma_2(y) \Rightarrow (\forall o \in \text{OutWrite}. (\forall 1 \vdash \sigma_1(o) = \sigma_2(o) \Rightarrow \text{ax} \sigma_1(o) = \text{ax} \sigma_2(o)))$
\item $\forall \sigma_1, \sigma_2. \sigma \rightarrow \sigma' \Rightarrow (\forall o \in \text{OutWrite}. (\forall 1 \vdash \sigma_1(o) = \sigma_2(o))$
\item $\forall \forall x \in \text{VarWrite}. \text{ax} \sigma_1(x) = \text{ax} \sigma_2(x) \Rightarrow \sigma \rightarrow \sigma'$
\end{enumerate}
An example of the second case, where an extra is produced, is the union of the dependencies of each branch. If both branches are executed, the dependencies are the value dependencies. For instance, for a conditional where both branches are executed, the dependencies are the union of the dependencies of each branch.

\[
\text{CD, } D, t \parallel \mu, D', VD' \quad \text{where}
\]

\[
VD' = CD \cup \text{InputRead} \bigcup_{x \in \text{VarRead}} D(x)
\]

\[
\forall x. D'(x) = \begin{cases} VD' & \text{if } x \in \text{VarWrite} \\ D(x) & \text{otherwise} \end{cases}
\]

\[
\forall o. D'(o) = \begin{cases} VD' \cup D(o) & \text{if } o \in \text{OutputWrite} \\ D(o) & \text{otherwise} \end{cases}
\]

Finally the Merge rule simply merges the dependencies together by doing the pointwise union of the dependencies environments, and the union of the value dependencies. For instance, for a conditional where both branches are executed, the dependencies are the union of the dependencies of each branch.

\[
\text{CD, } D, t \parallel \mu_1, D_1, VD_1 \quad \mu_1 = \text{upSome}(\text{fst}(\mu)) \circ \mu_1
\]

\[
\text{CD, } D, t \parallel \mu_2, D_2, VD_2 \quad \mu_2 = \text{nextSome}(\text{snd}(\mu_2)) \circ \mu_2
\]

The reader can easily verify that the first branch is derived as

\[
\begin{align*}
& \text{if Input 1} \\
& \text{then Output 1 0} \\
& \text{else Output 1 0}
\end{align*}
\]

Figure 10: A program for which the annotations will over-approximate

Note that annotations only approximates the notion of non-interference and we may capture dependencies that do not lead to interferences. For example, Figure 10 shows a program for which any annotated multiderivation will calculate that the output 1 depends on the input 1, although changing input 1 would not change the result. The loss of precision comes from the fact that we only track dependencies, and not the actual values being computed.

5.2 Capturing masking

In Figure 11, we have re-written the running example of Figure 4 in the WHILE language with the value of secret stored in the first input and the public variable being the first output. We now show how our approach captures the dependency.

\[
x := \text{true}; \quad y := \text{true};
\]

\[
\begin{align*}
& \text{if Input 1} \\
& \quad \text{then } x := \text{false} \\
& \quad \text{else skip;}
\end{align*}
\]

\[
\begin{align*}
& \text{if } x \\
& \quad \text{then } y := \text{false} \\
& \quad \text{else skip;}
\end{align*}
\]

\[
\text{Output 1 } y
\]

Figure 11: The running example in the WHILE language

Consider two states, one with false in the first input and one with true. We derive the running example in the annotated multi-semantics. We write D0 the empty dependencies environment, a function returning an empty set for every variable and output. We suppose x and y are already set to true and the dependencies are empty.

When evaluating the first if statement, we have to derive the condition Input 1 and then derive each branch with a smaller relation (after applying rule Merge) depending on the condition. The reader can easily verify that the first branch is derived as

\[
\text{[1], D}_0, \text{If, } x := \text{false skip} \Downarrow \mu_{\text{true}}, D_0[x \mapsto \{1\}, \{1\}]
\]

and second branch is derived as

\[
\text{[1], D}_0, \text{If, } x := \text{false skip} \Downarrow \mu_{\text{false}}, D_0, [1]
\]

where \( \mu_{\text{true}} \) and \( \mu_{\text{false}} \) are the singleton multistates relating only the corresponding states in the Pretty-Big-Step semantics for both derivation.
where $\mu = \mu_{\text{true}} \cup \mu_{\text{false}}$

Putting these together, the derivation of $P_1 = \text{if Input 1 then } x := \text{false else skip}$ is

$$\emptyset, D_0, \text{Input 1} \Downarrow \mu_{\text{input}}, D_0, (1)$$

$$\emptyset, D_0, P_1 \Downarrow \mu_{P_1}, D_0, (1)$$

where $\mu_{\text{input}}$ and $\mu_{P_1}$ are the multistates each one relating two pairs of states corresponding the derivations in the Pretty-Big-Step semantics for the terms.

Without even going further, we already know that $x$ depends on the first input. The second if statement has the same behavior: at the end we also infer that $y$ depends on the first input.

Finally, when observing $y$, the dependency flows into the first output. If we call our program runningExample we have

$$\emptyset, D_0, \text{runningExample} \Downarrow \mu_{\text{RE}}, D_0, (1)$$

where $D = D_0[x \mapsto (1)][y \mapsto (1)][1 \mapsto (1)]$ and $\mu_{\text{RE}}$ is the relation relating the two pairs of states appearing in the corresponding Pretty-Big-Step derivations. We can observe that we have $1 \in D(1) = (1)$.

### 5.3 Precision

As our framework relies on executions, we can potentially be more precise than static analyses. This is not surprising as we do not provide analyses, but a way to prove their correction. Thus, very precise analyses that can infer which branch of a conditional is taken can still be proven correct with our framework.

To illustrate this, we suppose that our language has been extended with the infix operators $\times$, $\div$, $+$, which are respectively the lower or equal operator, the equal operator and the addition operator. We also introduce the logical not operator and we use a shortcut $\text{isprime}$ to represent an expression returning $true$ if $i$ is prime and false otherwise (for the purpose of this example, it could just be a disjunction of equalities between $i$ and all of the prime numbers smaller than 200). In the example of Figure 12, in any annotated multiderivation, the annotations will show that $x$ does not depend on input 1 because in every execution, the loop will end up overwriting the value of $x$ by the constant 0. It implies that output 1 depends on nothing. In the other hand, a syntactic method (for example we could adapt one from Sabelfeld and Myers approach [19]) approximates the dependencies after the if statement saying that $x$ depends on input 1, and then the output 1 also depends on input 1.

Let $t$ the program of Figure 12. We have the following result.

**Lemma 5.1.** For every $\mu$, if $\emptyset, D_0, t \Downarrow \mu, D', V \cup D'$, then $D'(1) = \emptyset$.

### 6 Correctness

We now formally prove that the framework is correct.

#### 6.1 Correctness theorem

We define $\Delta(\sigma_1, \sigma_2)$ as the set of variables and outputs on which the two states differ.

**Definition 6.1.** Let $\sigma_1$ and $\sigma_2$ be two states.

$$\Delta(\sigma_1, \sigma_2) = \{ x \in \text{Var} | \sigma_1(x) \neq \sigma_2(x) \} \cup \{ o \in \text{Outputs} | \sigma_1(o) \neq \sigma_2(o) \}$$

Two derivations are said to be $(I, o)$-interferent if a difference in only the inputs $I \subseteq \text{Inputs}$ results in a difference in the output $o \in \text{Outputs}$.

**Definition 6.2.** Let $t$ be a term, $I$ a finite set of private inputs and $o$ a public output. $t$ is $(I, o)$-interferent if there exist four states $\sigma_1, \sigma'_1, \sigma_2$ and $\sigma'_2$ such that $\sigma_1, t \rightarrow \sigma'_1, \sigma_2, t \rightarrow \sigma'_2$ and:

- $\forall i \in I, \sigma_1(i) = \sigma'_1(i)$
- $\Delta(\sigma_1, \sigma_2) = \emptyset$
- $\text{extra}(\sigma_1) = \text{extra}(\sigma_2)$
- $\sigma'_1(o) \neq \sigma'_2(o)$

We will note this formula $\text{inter}_f(I, o)(\sigma_1, \sigma'_1, \sigma_2, \sigma'_2)$.

A term is then interferent if and only if there exists a finite set of inputs $I$ and an output $o$ such that it is $(I, o)$-interferent.
The fundamental theorem 6.3 is the main theorem we want to prove. It says that if we have two Pretty-Big-Step derivations showing that output $o$ depends on inputs $I$, then their exists an annotated multiderivation with empty dependencies on the left such that the annotation shows this interference. By contraposition it means that if for every multiderivation we cannot show interference by the annotations, then the program is non-interferent.

**Theorem 6.3 (Fundamental Theorem).**  $\forall t, \sigma_1, \sigma_1', \sigma_2, \sigma_2', I, o.$  
  \[
  \sigma_1.t \rightarrow \sigma_1' \land \sigma_2.t \rightarrow \sigma_2' \land \text{interf}_{(I,o)}(\sigma_1, \sigma_1', \sigma_2, \sigma_2') \implies \exists \mu, D', VD' \text{ such that } \\
  \emptyset, D_0, t \downarrow \mu, D', VD' \land (\sigma_1, \sigma_1') \in \mu \land (\sigma_2, \sigma_2') \in \mu \land I \subseteq D'(o)
  \]

The fundamental theorem is a particular case of the more general lemma A.1 when we take $CD = \emptyset$ and $D = D_0$.

6.2 Proving an analysis

Given a program, proving the absence of information leakage with this framework would require considering every annotated multiderivation with exactly two pairs of states in the multistate and prove that there is no unwanted dependency. But proving interference requires only one annotated multiderivation. This allows us to use the framework to prove analyses.

Let us consider an analysis $A$. It is a function returning $true$ for at least each interferent program and may have some false-positives. But if the function returns $false$, it means the analyzed program satisfies the property of non-interference.

The standard way to prove the analysis $A$ is the following:

**Lemma 6.4.**  
  $\forall P$,  
  if $P$ is interferent then $A(P)$

Such proofs are difficult to do by induction of the program since non-interference is an hyperproperty that is not defined by induction. When assuming the hypothesis "$P$ is interferent", we only have information on what happens before two executions (the states differ only on some private inputs) and after (the resulting states differ on a public output). No information is given on what happens in the program.

Instead, if one uses our framework, he has to prove:

**Lemma 6.5.**  
  $\forall P, I, o$  
  if $CD, D, P \not\downarrow \mu, D', VD' \land I \subseteq D'(o) \land I$ are private $\land o$ is public then $A(P)$

because if $P$ is interferent then the hypothesis of lemma 6.5 is satisfied for some $I$ and $o$. This proof can be done by induction on the annotated multiderivations. It is easier to manipulate because assuming that we have a leaking annotated derivation gives us a whole derivation tree with annotations at each semantic step.

The drawback of our approach is that one cannot prove the correctness of an analysis that is more complete than our method.

7 RELATED WORK


One groups users, using a certain set of commands, is noninterfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see.

There are several modern definitions of non-interference. In particular, non-interference may take into account the termination of an execution of the program. We thus have termination-insensitive non-interference [1], termination-aware non-interference [6], and timing- and termination-sensitive non-interference [16]. Our work considers termination-insensitive non-interference. To be able to deal with non-terminating executions, we would need to consider a coinductive version of the semantics.

A major inspiration of our work is the 2003 paper by A. Sabelfeld & A. C. Myers [19]. They give an overview of the information-flow techniques and show the many sources of potential interference. Our long-term goal is to evaluate our approach with the full Pretty-Big-Step semantics of JavaScript [7] and to show that [19] listed every possible source of information leak.

The thesis of G. Le Guernic [17] proposes and proves a precise dynamic analysis for non-interference. T. Austin and C. Flanagan also propose sound dynamic analyses for non-interference based on the no-sensitive-upgrade policy [2] and the permissive upgrade policy [3]. Our approach is similar in the sense that it is based on actual executions, but we consider every execution whereas these works monitor a single execution, modifying it if it is interferent. We believe, and should prove, that we are at least as precise as these works. Our goals are also quite different: they provide a monitor, we provide a framework to simplify the certification of analyses.

A. Sabelfeld and A. Russo [18, 20] prove several properties comparing static and dynamic approaches of non-interference. In particular, purely dynamic monitors can not be sound and permissive but it is possible for an hybrid monitor. Our framework could be a way to certify the correctness of such hybrid monitors.

G. Barthe, P.R. D’Argenio & T. Rezk [5] reduce the problem of non-interference of a program into a safety property of a transformation of the program. It allows to use standard techniques based on program logic for information flow verification. Our work is similar in the sense that we both transform a hyperproperty into a property. Self-composition achieves it by transforming the program, whereas we achieve it by extending the semantics in a mechanical way. In addition, our approach never inspects the values produced by the program, but only how it manipulates them. This is the reason why our approach is incomplete. For instance, we do not identify when two branches of a conditional do the same thing and we may flag it as interferent.

S. Hunt & D. Sands [15] present a family of semantically sound type system for non-interference. The main relation between the paper is the use of dependencies: a mapping from a variable to sets of variables they depend on in [15], a mapping from variables and outputs to set of inputs in our case. Our work is more precise as it does not use program points but actual executions. We also never consider the dependencies from branches of conditionals that are
taken by no execution, as illustrated in Figure 12. Finally, we do not propose an analysis, but a generic way to mechanically build the refined semantics.

D. Devriese and F. Piessens [13] introduce the notion of secure multi-execution allowing a sound and precise technique for information flow verification by execution a program multiple times with different security levels. Inspired by this work, T. Austin and C. Flanagan [4] present a new dynamic analysis for information flow based on faceted values. Our approach lies between secure multi-execution and faceted execution: we do not tag data but spawn multiple executions. In our pretty-big-step setting, however, the continuations of those executions are shared, in a way reminiscent of faceted execution.

8 CONCLUSION

In this paper, we presented a framework to automatically refine a semantics written in Pretty-Big-Step form into a new multisemantics able to consider many derivations at once and proved with the Coq proof assistant a correctness relation between the new and old semantics. We then presented an extension of the multisemantics with annotations that soundly approximates the notion of non-interference. The correctness proofs of the annotations is done by hand in the appendices. The final annotated multisemantics is a tool to prove the correctness of non-interference analyses.

Our next step is the full proof in Coq of the approach, followed by the extensions of the example language to show we can capture information flows in presence of functions and exceptions. We then want to apply the approach to certify existing analyses. Finally, we plan to refine the annotations in the Merge rule to inspect the results of computation, only adding dependencies when the results differ. We conjecture this will result in a framework that is complete in relation to non-interference.

REFERENCES


Appendices

Before starting the proofs, we recall important hypothesis over the Pretty-Big-Step semantics:

- For a given term and a given extra, there is a most one rule derivable,
- The functions \( \text{up} \) don’t change the memory but only the extra.
- The functions \( \text{next} \) keep the memory of the second argument and the extra depends only on the extras of the arguments.

A PROOF OF CORRECTNESS OF THE ANNOTATIONS

**Lemma A.1.** \( \forall t, \sigma_1, \sigma_1', \sigma_2, \sigma_2', l. \)
\[ \sigma_1, t \rightarrow \sigma_1' \]
\[ \sigma_2, t \rightarrow \sigma_2' \]
\[ \Rightarrow \forall i' \in \text{Inputs} \land \sigma_1(i') = \sigma_2(i') \]
\[ \forall i \in I, \sigma_1(i) \neq \sigma_2(i) \]
\[ \Rightarrow \forall D, CD, \]
\[ \forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x) \]
\[ \Rightarrow (\text{extra}(\sigma_1) \neq \text{extra}(\sigma_2) \Rightarrow I \subset CD) \]

\[ \exists D', \exists \sigma_1', \exists \sigma_2', \exists \sigma_1'', \exists \sigma_2'', \exists I' \]
\[ \exists \sigma_1', \exists \sigma_2', \exists \sigma_1'', \exists \sigma_2'', \exists I' \]
\[ \forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x) \]
\[ \Rightarrow (\text{extra}(\sigma_1') \neq \text{extra}(\sigma_2') \Rightarrow I \subset CD) \]

**Proof.** Let \( t, \sigma_1, \sigma_1', \sigma_2, \sigma_2', l. \)

Let \( \sigma_1, t \rightarrow \sigma_1' \) and \( \sigma_2, t \rightarrow \sigma_2' \) be two Pretty-Big-Step derivations. Let’s continue the proof by induction on the first derivation and then by a case matching on the second one.

**First case.** In the case of the two different rules \( R \) and \( R' \), we necessarily have \( \text{extra}(\sigma_1) \neq \text{extra}(\sigma_2) \) by hypothesis on the Pretty-Big-Step semantics. We will have to use the Merge rule.

Let suppose

\[ \forall i' \in \text{Inputs} \land \sigma_1(i') = \sigma_2(i') \]
\[ \forall i \in I, \sigma_1(i) \neq \sigma_2(i) \]

Let’s have \( D \) and \( CD \) such that

\[ \forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x) \]
\[ (\text{extra}(\sigma_1) \neq \text{extra}(\sigma_2) \Rightarrow I \subset CD) \]

By this last hypothesis we have \( I \subset CD \). Thanks to lemma B.1 there exists \( D'_1, VD'_1, D'_2, VD'_2 \) such that \( CD, D, t \not\leq\{\{\sigma_1, \sigma_2, \sigma_2', \sigma_2''\}\}, D', VD' \)
and \( CD, D, t \not\leq\{\{\sigma_1', \sigma_2', \sigma_2'', \sigma_2''\}\}, D'', VD'' \).

We can construct:

- \( VD' = VD'_1 \cup VD'_2 \)
- \( D'(xo) = D'_1(xo) \cup D'_2(xo) \)

We now have our 3 points:

1. \( CD, D, t \not\leq\{\{\sigma_1', \sigma_2', \sigma_2'', \sigma_2''\}\}, D', VD' \) thanks to the merge rule
2. \( \forall y \in \Delta(\sigma_1', \sigma_2') \), either, \( y \in \Delta(\sigma_1, \sigma_2) \) and then by hypothesis, \( I \subset D(y) \).
   Lemma B.2 ensures that \( I \subset D'(y) \).
   or, \( \sigma_1(y) = \sigma_2(y) \) and we can assume (for symmetric reasons) that \( \sigma_1(y) \neq \sigma_2(y) \). Lemma B.3 ensures that \( CD \subset D'(y) \) and thus \( I \subset CD \subset D'(y) \).
   Anyway, \( I \subset D'(y) \).

- **Second case.** In the other case, both derivation is made by the same rule:

  - **Axiom:**
    \[ \text{Ax} \sigma_1, t \rightarrow \sigma_1' \]
    and,
    \[ \text{Ax} \sigma_2, t \rightarrow \sigma_2' \]

    Let suppose
    \[ \forall i' \in \text{Inputs} \land \sigma_1(i') = \sigma_2(i') \]
    \[ \forall i \in I, \sigma_1(i) \neq \sigma_2(i) \]

    Let’s have \( D \) and \( CD \) such that
    \[ \forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x) \]
    \[ (\text{extra}(\sigma_1) \neq \text{extra}(\sigma_2) \Rightarrow I \subset CD) \]

    We can construct:
    - \( VD' = CD \cup \text{InputRead} \)
    - \( D'(xo) = \begin{cases} \text{InputWrite} & \text{if } xo \in \text{VarWrite} \\ \text{OutputWrite} & \text{if } xo \in \text{VarWrite} \\ \text{varWrite} & \text{otherwise} \end{cases} \)

    We now have 3 points to prove:

    1. By construction of \( D' \) and \( VD' \), we have the derivation
       \( D, CD, t \not\leq\{\{\sigma_1, \sigma_1', \sigma_2, \sigma_2'\}\}, D', VD' \)

    2. \( \forall y \in \Delta(\sigma_1', \sigma_2') \), we have 4 cases:
       - **\( y \) is a variable and \( y \in \text{VarWrite} \).**
         By hypothesis on the elements of \( \text{VarWrite} \), \( \text{extra}(\sigma_1) \neq \text{extra}(\sigma_2) \), \( (\exists i' \in \text{InputRead}, \sigma_1(i') = \sigma_2(i')) \), or \( (\exists x \in \text{VarWrite}, \sigma_1(x) \neq \sigma_2(x)) \). We can simplify this into:
           - **\( \sigma_1(y) \neq \sigma_2(y) \):** \( (\exists x \in \text{VarWrite}, x \in \Delta(\sigma_1, \sigma_2)) \).
           - **\( \sigma_1(x) \neq \sigma_2(x) \):** \( (\exists x \in \text{VarWrite}, x \in \Delta(\sigma_1, \sigma_2)) \). It means that either
             - \( I \subset CD \), \( I \subset \text{InputRead} \), or \( I \subset \text{VarWrite} \).
             - **In the three cases, \( I \subset D' \), and thus \( I \subset D'(y) \).**
           - **\( y \) is a variable and \( y \notin \text{VarWrite} \).**
             By hypothesis on the elements not member of \( \text{VarWrite} \), \( \sigma_1(y) \neq \sigma_2(y) \) i.e. \( y \in \Delta(\sigma_1, \sigma_2) \) and by hypothesis, \( I \subset D(y) \).
             - **\( y \) is an output and \( y \notin \text{OutputWrite} \).**
               Let’s define \( l_1 \) and \( l_2 \) such that \( \sigma_1(y) = l_1 : \sigma_1(y) \) and \( \sigma_2(y) = l_2 : \sigma_2(y) \). Since \( l_1 : \sigma_1(y) \neq l_2 : \sigma_2(y) \), either \( l_1 \neq l_2 \) or \( y \in \Delta(\sigma_1, \sigma_2) \).
               With the same reasoning than for the first case: either \( I \subset CD \), \( I \subset \text{InputRead} \), \( (\exists x \in \text{VarWrite}, I \subset D(x)) \), or \( I \subset D(y) \). We have in every cases: \( I \subset D'(y) \).
             - **\( y \) is an output and \( y \notin \text{OutputWrite} \).**
               By hypothesis on the elements not member of \( \text{OutputWrite} \), \( \sigma_1(y) \neq \sigma_2(y) \) i.e. \( y \in \Delta(\sigma_1, \sigma_2) \) and by hypothesis, \( I \subset D(y) \).
               - **R1:**
                 \[ \text{R1} \sigma_1, t \rightarrow \sigma_1' \]

- **R1:**
  \[ \text{up}(\sigma_1, t) \rightarrow \sigma_1' \]
Let 
\(\forall i \in Inputs[I, \sigma_1(i')] = \sigma_2(i')\)

\(\forall i \in I, \sigma_1(i) \neq \sigma_2(i)\)

Let’s have \(D\) and \(CD\) such that 
\(\forall x \in \Delta(\sigma_1, \sigma_2), \sigma_1(x) \in D(x)\)

\((extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)\)

Since \(up\) does not modify the memory, to use the induction hypothesis we only need to prove 
\(extra(up(\sigma_1)) \neq extra(up(\sigma_2)) \implies I \subset CD\)

Which is a consequence of \((extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)\).

So by induction hypothesis: \(3D'_1, VD'_1\) such that

1. \(CD, D, t \cup ((up(\sigma_1), \sigma_1'), (up(\sigma_2), \sigma_2')), D'_1, VD'_1\)
2. \(\forall y \in \Delta(\sigma_1, \sigma_2), I \subset D'(y)\)
3. \(extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD'_1\)

We can then have the 3 points:
1. \(CD, D, t \cup ((\sigma_1, \sigma_2), (\sigma_1, \sigma_2')), D'_1, VD'_1\)
2. \(\forall y \in \Delta(\sigma_1, \sigma_2), I \subset D'(y)\)
3. \(extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD'_1\)

\(\because \)
\(R_2:\)
\(up(\sigma_1), t_1 \rightarrow \sigma_1''\)
\(next(\sigma_1, \sigma_1'), t_2 \rightarrow \sigma_1'\)

and,
\(up(\sigma_2), t_1 \rightarrow \sigma_2''\)
\(next(\sigma_2, \sigma_2'), t_2 \rightarrow \sigma_2'\)

\(\because \)
\(R_1:\)
\(up(\sigma_2), t_1 \rightarrow \sigma_2''\)
\(next(\sigma_1, \sigma_1'), t_2 \rightarrow \sigma_1'\)

Let suppose
\(\forall i' \in Inputs[I, \sigma_1(i')] = \sigma_2(i')\)

\(\forall i \in I, \sigma_1(i) \neq \sigma_2(i)\)

Let’s have \(D\) and \(CD\) such that
\(\forall x \in \Delta(\sigma_1, \sigma_2), I \subset D(x)\)

\((extra(\sigma_1) \neq extra(\sigma_2) \implies I \subset CD)\)

Since \(up\) does not change the memory but only the extra and 
\(extra(up(\sigma_1)) \neq extra(up(\sigma_2)) \implies I \subset CD\), we can use the induction hypothesis: \(3D'_1, VD'_1\) such that

1. \(CD, D, t \cup ((up(\sigma_1), \sigma_1'), (up(\sigma_2), \sigma_2')), D'_1, VD'_1\)
2. \(\forall y \in \Delta(\sigma_1, \sigma_2), I \subset D'(y)\)
3. \(extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD'_1\)

Since the result of \(next\) has the same memory as the second argument (but not necessarily the same extra), we have

\(\Delta(next(\sigma_1, \sigma_1'), next(\sigma_2, \sigma_2')) = \Delta(\sigma_1', \sigma_2')\).

To use again the induction hypothesis on the second premise, we only need to prove
\(extra(next(\sigma_1, \sigma_1')) \neq extra(next(\sigma_2, \sigma_2')) \implies I \subset CD'\)

Where
\(CD' = CD \cup \begin{cases} VD'_1 & \text{if } \text{prod\_extra}, \\ 0 & \text{otherwise.} \end{cases}\)

If \(extra(next(\sigma_1, \sigma_1')) \neq extra(next(\sigma_2, \sigma_2'))\) then either \(extra(\sigma_1) \neq extra(\sigma_2)\) and then by hypothesis \(I \subset CD\), or \(extra(\sigma_1') \neq extra(\sigma_2')\) and then \(t_1\) produces an extra and by hypothesis \(I \subset VD'_1\). In both case, \(I \subset CD'\)

We can then use the induction hypothesis another time: \(3D'_2, VD'_2\) such that

1. \(CD', D_1, t_2 \cup ((next(\sigma_1, \sigma_1'), \sigma_1'), (next(\sigma_2, \sigma_2'), \sigma_2')), D'_2, VD'_2\)
2. \(\forall y \in \Delta(\sigma_1', \sigma_2'), I \subset D'(y)\)
3. \(extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD'_2\)

We finally have:
1. \(CD, D, t \cup ((\sigma_1, \sigma_1'), (\sigma_2, \sigma_2'), D'_2, VD'_2\)
2. \(\forall y \in \Delta(\sigma_1', \sigma_2'), I \subset D'(y)\)
3. \(extra(\sigma_1') \neq extra(\sigma_2') \implies I \subset VD'_2\)

\(\square\)

B OTHER LEMMAS

The following lemma states that if we have a Pretty-Big-Step derivation, then we can build an annotated multiderivation from it.

**Lemma B.1.** \(\forall \sigma, \sigma', t, \sigma, t \rightarrow \sigma' \implies \forall CD, D, \exists CD', D', VD'

\(CD, D, t \cup \mu, D', VD'\)

\(\implies \forall I, xo, \)

\(I \subset CD\)

\(I \subset CD'\)

\(( I \subset D(xo) \implies I \subset D'(x) )\)

**Proof.** Straightforward by induction since the condition needed by every pair of states related by a \(\mu\) in a multisemantics rule is exactly the condition verified by the pair of state in the corresponding Pretty-Big-Step rule.

\(\square\)

**Lemma B.2** states that if before a multiexecution the context depends on inputs \(I\) then the calculated value will also depend on \(I\), and moreover, if a variable or an output \(xo\) also depends on \(I\) then \(xo\) will depend on \(I\) at the end of the execution.

**Lemma B.2.** \(\forall CD, D, t, \mu, D', VD'

\(CD, D, t \cup \mu, D', VD'\)

\(\implies \forall I, xo, \)

\(I \subset CD\)

\(I \subset CD'\)

\(( I \subset D(xo) \implies I \subset D'(x) )\)

**Proof.** Let have \(CD, D, t, \mu, D', VD'\) such that we have the multi-derivation \(CD, D, t \cup \mu, D', VD'\) and prove the lemma by induction on this derivation.

**Axiom**

\[ A_x(t) \quad \mu = ax\text{Same}(\text{fst}(\mu)) \quad \mu \neq \emptyset \]

**With**

\[ VD' = CD \cup \text{InputRead} \quad \cup \quad \text{D'(x)} \quad \text{x in VarRead} \]

\[ \forall x, D'(x) = \begin{cases} VD' & \text{if } x \in \text{VarWrite} \\ D(x) & \text{otherwise} \end{cases} \]

\[ \forall o, D'(o) = \begin{cases} VD' \cup D(o) & \text{if } o \in \text{OutputWrite} \\ D(o) & \text{otherwise} \end{cases} \]

Let’s have \(xo\) a variable or an output and \(I \subset CD\). We directly have \(I \subset CD'\).

Moreover if \(I \subset D(xo)\), whether \(xo\) in VarWrite, \(xo \notin\) VarWrite, \(xo \in\) OutputWrite or \(xo \notin\) OutputWrite, we have \(I \subset D'(x)\).
During an execution, then

Let have

Lemma B.3. \( \forall \sigma, \sigma', xo.\)

\[ (\sigma, \sigma') \in \mu \]

\[ \Rightarrow \sigma(xo) \neq \sigma'(xo) \]

\[ \Rightarrow CD \subset D'(xo) \]

Proof. Let have \( CD, D, t \parallel \mu, D', VD' \) such that we have the multi-derivation \( CD, D, t \parallel \mu, D', VD' \) and prove the lemma by induction on this derivation.

Rule 1

\[
\begin{align*}
CD, D, t_1 \parallel \mu_1, D', VD' \\
R_{1(t)}(t) \\
\mu = u\text{p}_{\text{Some}}(\text{f}st(\mu)) \circ \mu_1 \\
CD, D, t \parallel \mu, D', VD'
\end{align*}
\]

Let's have \( xo \) a variable or an output and \( I \subset CD \).

By induction hypothesis we directly have the result:

\[ I \subset VD' \land (I \subset D(xo) \Rightarrow I \subset D'(xo)) \]

Rule 2

\[
\begin{align*}
CD, D, t_1 \parallel \mu_1, D_1, VD_1 \\
R_{2(t)}(t) \\
\mu = \mu_n \circ \text{next}_{\text{Some}}(\text{snd}(\mu_n)) \circ \mu_2 \\
CD', D_1, t_2 \parallel \mu_2, D', VD'
\end{align*}
\]

With

\[
CD' = CD \cup \{VD_1 \text{ if } \text{prod}_{\text{extra}} \}
\]

\[ \theta \text{ otherwise} \]

Let's have \( xo \) a variable or an output and \( I \subset CD \).

By induction hypothesis on the first premise we have:

\[ I \subset VD_1 \land (I \subset D(xo) \Rightarrow I \subset D_1(xo)) \]

and by induction on the second:

\[ I \subset VD' \land (I \subset D_1(xo) \Rightarrow I \subset D'(xo)) \]

By combining both implication:

\[ I \subset VD' \land (I \subset D(xo) \Rightarrow I \subset D'(xo)) \]

which is the result we wanted.

Merge

\[
\begin{align*}
CD, D, t \parallel \mu_1, D_1, VD_1 \\
\text{Merge}(t) \\
CD, D, t \parallel \mu_2, D_2, VD_2 \\
\mu = \mu_1 \cup \mu_2 \\
CD, D, t \parallel \mu, D', VD'
\end{align*}
\]

With

\[
VD' = VD_1 \cup VD_2
\]

\( \forall xo \in \text{Var} \cup \text{Out}, D'(xo) = D_1(xo) \cup D_2(xo) \)

Let's have \( xo \) a variable or an output and \( I \subset CD \).

By induction hypothesis on both premises we have

\[ I \subset VD_1 \land (I \subset D(xo) \Rightarrow I \subset D_1(xo)) \]

and

\[ I \subset VD_2 \land (I \subset D(xo) \Rightarrow I \subset D_2(xo)) \]

Thus,

\[ I \subset VD' \land (I \subset D(xo) \Rightarrow I \subset D'(xo)) \]

We have the result by induction. \( \square \)

This lemma states that if a variable or an output \( xo \) is modified during an execution, then \( xo \) depends at least on the context of the execution.

Lemma B.3. \( \forall CD, D, t, \mu, D', VD' \)

\[
\begin{align*}
CD, D, t \parallel \mu, D', VD' \\
\Rightarrow \forall \sigma, \sigma', xo. \\
(\sigma, \sigma') \in \mu \\
\Rightarrow \sigma(xo) \neq \sigma'(xo) \\
\Rightarrow CD \subset D'(xo)
\end{align*}
\]

Proof. Let have \( CD, D, t, \mu, D', VD' \) such that we have the multi-derivation \( CD, D, t \parallel \mu, D', VD' \) and prove the lemma by induction on this derivation.
∀ o, D'(o) = D₁(o) ∪ D₂(o)

Let's have two states σ and σ' such that (σ, σ') ∈ μ and a variable or an output xo such that σ(xo) ≠ σ'(xo).

For symmetric reason, we can suppose (σ, σ') ∈ μ₁ and thus the induction hypothesis ensures CD ⊂ D'_1(xo) ⊂ D'(xo).