HAL
open science

Non Local Analyses Certification With an Annotated Semantics

Gurvan Cabon

To cite this version:

Gurvan Cabon. Non Local Analyses Certification With an Annotated Semantics. Programming Languages [cs.PL]. Université Rennes 1, 2018. English. NNT : . tel-01978292v1

HAL Id: tel-01978292

https://inria.hal.science/tel-01978292v1

Submitted on 11 Jan 2019 (v1), last revised 25 Feb 2019 (v2)

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.

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique
Par

« Gurvan CABON »

« Non Local Analyses Certification With an Annotated Semantics »

Thèse présentée et soutenue à RENNES, le 14 décembre 2018
Unité de recherche : Inria

Rapporteurs avant soutenance :

Tamara REZK, INRIA Sophia Antipolis-Méditerranée

Daniel HIRSCHKOFF, LIP Lyon

Composition du jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition ne comprend que les membres présents

Examinateurs : Sophie PINCHINAT, INRIA Rennes
Arthur CHARGUERAUD, Iniria & ICUBE, Université de Strasbourg
Sylvain CONCHON, LRI Paris-Sud

Dir. de thèse : Alan SCHMITT, INRIA Rennes

ACKNOWLEDGEMENT

- Alan - Jury - Celtique - famille - Profs - amis / copine - Batman et Bilou

TABLE OF CONTENTS

Résumé en français **5**

Introduction **9**

1 Background **13**
1.1 Non-interference 13
1.1.1 Definition 14
1.1.2 Undecidability 14
1.1.3 Flow patterns 15
1.1.4 Conclusion 17
1.1.5 Related work 18
1.2 Coq proof assistant 21
1.2.1 A proof assistant 21
1.2.2 Type classes 23
1.2.3 Partial functions 25
1.3 Pretty-Big-Step 25
1.3.1 What is PBS ? 26
1.3.2 Advantages 29
1.3.3 A While language in PBS 33

2 Generic format of semantics **39**
2.1 Motivation 39
2.2 Formal Pretty-Big-Step 39
2.2.1 Abstract PBS 39
2.2.2 Concretization of the WHILE language 43
2.3 Conclusion 50

3 Multisemantics **51**
3.1 Preliminary definitions 51
3.2 Canonical structure 54

3.3 Properties 58
3.4 Multisemantics of WHILE 59
3.5 Conclusion 63

4 Annotations **67**
4.1 Construction 67
4.2 Annotated multisemantics of WHILE 73
4.3 Capturing masking 78
4.4 Limits 81

5 Correctness **83**
5.1 Hypotheses 83
5.2 Correctness theorem 89
5.3 Proving an analyzer 91

Conclusion **95**

Bibliography **97**

A PBS rules of WHILE in Coq **105**

RÉSUMÉ EN FRANÇAIS

Des ordinateurs aux équipements médicaux en passant par les montres et les voitures, depuis plusieurs décenies les logiciels occupent une place de plus en plus importante dans tous les aspects de notre vie. Ils peuvent nous assister dans notre quotidien (appareils ménagers, voitures, etc.), nous permettre de communiquer avec les autres grâce à des outils connectés (ordinateurs, smartphones, montres, etc.) et nous divertir par le biais de jeux, de films et de musique.

Ces changements ont certes eu des effets bénéfiques sur notre mode de vie dans de nombreux domaines : santé, transports, science, agriculture, etc., mais il ne faut pas oublier que les logiciels peuvent ne pas avoir le comportement exact que nous attendons en raison de la complexité des mécanismes mis en place. Ces soi-disant bogues peuvent avoir des conséquences critiques : ils peuvent soit être la cause d'accidents, comme ce fut le cas pour l'échec du lancement d'Ariane 5 en 1996 [31], soit être une faille pour des attaques malveillantes telles que le bogue Heartbleed apparu en 2012 et divulgué en 2014 [1] permettant à un attaquant d'écouter et de voler des données de communications cryptées.

Méthodes formelles

Pour surmonter les bugs, les informaticiens ont d'abord mis au point des méthodes de test dans le but de détecter les comportements erronés. Les techniques de test présentent l'avantage que, lorsqu'un bogue est détecté, il est généralement possible de retracer la trace d'éxécution et de renvoyer automatiquement les informations nécessaires à sa résolution. Cependant, cela présente un inconvénient : ne pas trouver un bogue dans un programme ne garantit pas que celui-ci n'en comporte pas, car dans le cas général, le nombre des exécutions possibles est infini.

Inversement, les méthodes formelles permettent de prouver des propriétés et d'assurer qu'un programme ne présente pas certains comportements.

Contrairement aux méthodes de test, elles garantissent qu'aucun bogue lié à la propriété prouvée ne peut apparaître lors de l'exécution du programme analysé. Un autre avantage des méthodes formelles est qu'elles ne nécessitent pas l'exécution du programme et reposent sur des outils purement mathématiques.

Bien que les tests aient été largement utilisés dans l'industrie depuis de nombreuses décennies, les méthodes formelles sont maintenant suffisamment développées pour rivaliser avec les tests sur des projets de niveaux industriels : depuis 2011, Amazon utilise des spécifications formelles et des vérifications de modèles pour résoudre des problèmes concrets [32].

Vie privée et protection des données

Les logiciels étant partout, cela signifie qu'ils ont accès à de plus en plus de données. Combiné au fait que ces outils ont souvent la capacité de diffuser des données vers d'autres périphériques (via Internet par exemple), cela soulève une question de confidentialité importante. Il est souvent nécessaire de fournir une certaine quantité de donnée aux logiciels pour leur bon fonctionnement mais nous voudrions que ces données ne soient pas envoyés n'importe où. Par exemple, une personne qui se connecte à son compte bancaire à partir d'un navigateur ne veut certainement pas que son mot de passe soit envoyé partout sur Internet. Inversement, certains logiciels ont pour but de recevoir des données et de ne pas les afficher à l'utilisateur. C'est le cas, par exemple, des bloqueurs de publicités essayant de bloquer les publicités et des outils de sécurité parentale visant à ne pas afficher de contenu sensible. Le point commun de ces deux problèmes est la gestion du flux de données dans les logiciels.

Non-interférence

Une façon de donner des garanties sur le flux de données est d'assurer la *Non-Interférence*. La non-interférence est une propriété de sécurité qui garantit l'indépendance de certaines sorties spécifiques par rapport à des entrées spécifiques d'un programme. Pour revenir à l'exemple de saisie d'un mot de passe dans un navigateur, nous voudrions garantir que les données envoyées à tous les sites internet autres que le site internet de la banque (les

sorties spécifiques) ne dépendent pas du mot de passe saisi dans le champ "mot de passe" de la page internet de la banque (l'entrée spécifique).

La non-interférence est une *hyperpropriété* [17] : pour donner un contre-exemple à la propriété il ne suffit pas de montrer une exécution particulière du programme (contraire au accès mémoire illégaux par exemple) mais en comparant les résultat d'au moins deux exécutions.

But

Le but de cette thèse est de fournir un cadre permettant de prouver des analyseurs de non-interférence. Etant donné un langage avec sa sémantique, nous visons à construire une sémantique alternative dans laquelle une interférence peut être détectée par une seule dérivation, permettant ainsi des preuves simples par induction sur de telles dérivations.

Considérer qu'une seule exécution de la sémantique d'origine n'est clairement pas suffisant pour déterminer si un programme est non interférent. Étonnamment, étudier chaque exécution indépendamment et collecter des informations sur les dépendances n'est toujours pas suffisant. Nous proposons donc une approche formelle qui construit, à partir de toute sémantique respectant une certaine structure, une multisémantique permettant de raisonner simultanément sur plusieurs exécutions. L'ajout d'annotations à cette multisémantique nous permet de capturer les dépendances entre les entrées et les sorties d'un programme.

Nous montrons que notre approche est correcte, c'est-à-dire que les annotations capturent correctement la non-interférence. Cela nous permet de certifier des analyses correctes sans nous fier à la propriété de non-ingérence, mais plutôt à la multisémantique annotée. Pour illustrer notre approche, nous présentons un petit langage While et sa sémantique, nous construisons sa multisémantique annotée et nous prouvons un analyseur de ce langage.

Plan

Le reste du manuscrit est rédigé en anglais et est séparé en chapitre de la manière suivante : Le chapitre 1 introduit les notions préliminaires : la non-interférence, l'assistant de preuve Coq et Pretty-Big-Step. Le chapitre 2

formalise Pretty-Big-Step en Coq et un exemple concret de language WHILE. Le chapitre 3 décrit comment construire automatiquement une multisémantique. Le chapitre 4 annote cette multisémantique et explique le fonctionnement des annotations. Enfin, le chapitre 5 prouve la correction des annotations et montre comment utiliser la multisémantique annotée pour prouver un analyseur.

INTRODUCTION

From computers to medical equipment via watches and cars, since a few decades software has been taking an increasing place in every aspect of our lives. It can help our personal routines (as in household appliance, cars, ...), communicate with others thanks to connected tools (as computers, smartphones, watches), and entertain us through games, films and music. This changes have certainly brought benefits in our way of life in many domains : health, transport, science, agriculture, ... but it must not be forgotten that software may not have the exact behaviour we expect because of the complexity of the mechanisms at stake. These so-called bugs may have critical consequences : they can either be the cause of accidents as it has been the case for the Ariane 5 launch failure in 1996 [31] or be a backdoor for malicious attacks as the Heartbleed bug allowing an attacker to eavesdrop and steal data from encrypted communications introduced in 2012 and disclosed in 2014 [1].

Formal methods

To overcome bugs, computer scientists firstly developed testing methods with the intent of finding wrong behaviours. Testing techniques have the advantage that, when a bug is found, it is generally possible to track back its source and automatically return the needed information to fix it. However this comes with a downside : not finding a bug in a program does not ensure that the program will always behave correctly since in the general case the set of all the possible executions is infinite.

Conversely, formal methods allow to prove properties and ensure that a program may not have some behaviours. Unlike testing methods, they give guarantees that no bug related to the proven property can appear during an execution of the analyzed program. Another advantage of formal methods is that they do not require to execute the program and rely on purely mathematical tools.

While testing has been widely used in the industry since many decades, formal methods are now developed enough to compete testing in industrial projects : since 2011, Amazon has used formal specification and model checking to solve real-life challenges [32].

Privacy and data protection

Software being everywhere, it implies that it has access to more and more data. Combined with the fact that those tools often have the ability to broadcast data to other devices (through the internet for example), it raises an important privacy issue. There is a lot of kind of data that we have to give to a software but at the same time that we don't want the software to send anywhere. For example, someone who logs in to his/her bank account from a browser certainly does not want his/her password to be send all over the internet. Conversely, there is software that has the purpose of receiving data an not display it to the user. For instance it is the case of ad-blockers trying to block ads and parental security tools that aim to not display sensitive content. The common point of these two issues is managing the data flow in software.

Non-interference

One way to give guarantees over the flow of data is to ensure *Non-Interference*. Non-interference is a program security property that gives guarantees on the independence of specific outputs from specific inputs of a program. Going back to the example of entering a password in a browser, we would want to guarantee that the data sent to every website that is not the bank's website (the specific outputs) does not depend on the password entered in the "password" field on the bank's web-page (the specific input).

Non-interference is a *hyperproperty* [17] : giving a counter-example to the property cannot be done by exhibiting one particular execution of the program (unlike illegal memory access for example), but by comparing the results of at least two executions.

Goal

The goal of this thesis is to provide a framework to prove analyzers of non-interference. When given a language with its semantics, we aim at building an

alternative semantics where interference may be detected through a single derivation, hence enabling simple proofs by induction on such derivations.

Considering a single execution of the original semantics is clearly not sufficient to determine if a program is non-interferent. Surprisingly, studying every execution independently and gathering dependency information is also not sufficient. We thus 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 lets us certify sound analyses without relying on the non-interference property, but relying instead on the annotated multisemantics. To illustrate our approach, we present a small While language and its semantic, we build its annotated multisemantics, and we prove an analyzer over this language.

Outline

Chapter 1 presents general background related to the non-interference property, the Coq proof assistant and Pretty-Big-Step. Chapter 2 gives the coq formalization of Pretty-Big-Step and a concrete WHILE language written in this formalization. Chapter 3 describes how to automatically build a multisemantics given a Pretty-Big-Step. Chapter 4 annotates the multisemantics and explains the annotation mechanism. Finally Chapter 5 proves the correctness of the annotations and shows how the framework is used in an analyzer proof.

Source code

Each theorem, lemma, hypothesis or definition that appears in the thesis and that has been formalized in coq comes with a symbol indicating a link to the online proofs scripts.

CHAPITRE 1

BACKGROUND

This first chapter focuses on giving some background on three main things : the non-interference property, the Coq proof assistant and the Pretty-Big-Step semantics.

In all the document we will follow some usual notations listed in Figure 1.1 in case of any ambiguity. The notations directly coming from our work are defined in the corresponding sections.

— For any function $f : E \to F$, and elements $x \in E$, $v \in F$,
$f[x \mapsto v]$ denotes the function $y \mapsto \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$.
— For any function $f : E \to F$, and subset $H \subset E$,
$f(H)$ is the image of H through f.
— For any partial function $f : E \to F$,
$Dom(f) \subseteq E$ denotes the domain on which f is defined ;
— The symbol :: is used for adding an element to a list and the symbol @ is used for list concatenation.

FIGURE 1.1 – Notations used in the whole document

1.1 Non-interference

Static analyses of non-interference take their roots in 1977 with E. Cohen [18] and D. E. Denning & P. J. Denning [21]. They both propose static methods to track the flow of sensitive data in a program. The property is then formalized in 1982 by J. A. Goguen & J. Meseguer [23] as following :

> One group of users, using a certain set of commands, is non-interfering 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.

This formalization oversteps the programming languages domain and can be applied in network security, social interactions, ...

1.1.1 Definition

In our case, we define non-interference specifically for programming languages. Suppose we have a programming language in which variables can be private or public. We say a program is *non-interferent* if, for any pair of execution that initially agree on the public variables, then the values of the public variables are the same after the execution. In other words, changing the value of the private variables does not influence the final 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 1 (Termination-Insensitive Non-interference - TINI)**.** *A program is* non-interferent *if, for any pair of* ***terminating*** *executions starting with* ***same value in the public variables****, the executions end with the* ***same value in the public variables****.*

This definition only considers finite program executions. We illustrate through four examples of increasing complexity where leaks of private information may happen and how one may detect them. In all of them `public` is a public variable and `secret` is a private variable.

If we look back at Goguen & Meseguer's definition, *the first group of users* is the private variables, *the second group* is the public variables and *the commands* are the terms of the language. It is important to notice that *what the second group can see* is the value in the public variables at the beginning and the end of an execution ; it does not includes what happens during the execution.

1.1.2 Undecidability

Before trying to capture non-interference, it is necessary to understand how hard it is and we show that it is undecidable. To prove that the non-interference property is undecidable, we reduce the problem of knowing if

a program is terminating-insensitive non-interferent to the halting problem which is known to be undecidable.

We suppose we have an algorithm $\mathcal{A}$ determining if a program is TINI or not and we build a new one solving the halting problem.

Let us consider a program P and an input I of the program. We also consider two variables $public$ and $private$ not appearing in P and I and that are respectively public and private. With these elements we build the program P' defined as

$$P(I); public := private$$

We claim that $P(I)$ halts if and only if P' is not TINI.

— On one hand, if $P(I)$ halts, then considering two executions (one with $true$ and the other one with $false$ in the variable $private$) will end up with different values in the variable $public$. This is ensured by the fact that the computation of $P(I)$ doesn't modify the value of $private$ since it doesn't appear in P and I. The definition of TINI doesn't hold for P'.

— On the other hand, if $P(I)$ doesn't halt, then the set of all terminating executions of P' is empty. Thus P' is TINI.

This way we can construct the following algorithm to solve the halting problem :

Input: P,I
if $\mathcal{A}(P(I); public := private)$ **then**
  return false
else
  return true
end

Thus, TINI is undecidable.

Knowing that TINI is undecidable makes impossible the existence of a complete and correct analysis.

1.1.3 Flow patterns

There are different ways to leak information and to make a program interferent (or not non-interferent). In order to introduce the initial idea behind the

multisemantics, we give a non-exhaustive list of program patterns that can leak information.

Direct flow As a simple first example, the program of Figure 1.2 is clearly interferent : changing the value of `secret` changes the value of `public`. If we stick to the definition, we can say this program is interferent because there exists two executions starting in the environments $\{public : true; private : false\}$ and $\{public : true; private : true\}$ (in which the values of the public variable are the same), and respectively ending in the environments $\{public : false; private : false\}$ and $\{public : true; private : true\}$ (in which the values of the public variable are different).

```
public := secret
```

FIGURE 1.2 – Direct information flows

This is a *direct flow* of information because the value of `secret` is directly assigned into `public`.

Indirect flow Unfortunately, direct flows are not the only sources of interference. It may also come from the context in which a particular instruction is executed. For example, the program of Figure 1.3 shows a program with an *indirect flow*. The value of `secret` is not directly stored into `public` but the condition in the if statements 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 this program would then witness the interference.

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

FIGURE 1.3 – Indirect information flows

Masking Another source of interference is the fact that *not* executing a part of the code can provide information. This is often called *masking*. The program of Figure 1.4 illustrates the phenomenon. 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.

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

FIGURE 1.4 – Masking

Double masking Unfortunately again, this is not sufficient. In this last example, in Figure 1.5, we can see that there exists no single execution where the flow can be inferred. We will refer to this example as the running example. In the case $\texttt{secret} = true$, `public` depends on `y`, which is not modified by the execution. In the other case $\texttt{secret} = false$, `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 $\texttt{public} = \texttt{secret}$ at the end of both execution, so the secret is leaked. Looking at every execution *independently* is not enough.

1.1.4 Conclusion

To retrieve the interference 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 the last example, 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`, because not modifying `x`

```
x := true
y := true
if secret
  then x := false
  else skip
if x
  then y := false
  else skip
public := y
```

FIGURE 1.5 – Double masking

leaks information about `secret`. We can similarly deduce that `y` depends on `x` in both executions, hence `public` transitively depends on `secret`.

We thus propose to transform 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.

$secret = true$

```
x := true
y := true
if secret
  then x := false
  else skip
if x
  then y := false
  else skip
public := y
```

$public = true$

$secret = false$

```
x := true
y := true
if secret
  then x := false
  else skip
if x
  then y := false
  else skip
public := y
```

$public = false$

(executed code, ~~non-executed code~~)

FIGURE 1.6 – Running Example

1.1.5 Related work

A major inspiration of this work is the 2003 paper by A. Sabelfeld & A. C. Myers [35]. 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 PBS semantics of JavaScript [13] and to show that Sabelfeld and Myers listed every possible source of information leak.

Dynamic analyses

The first dynamic mechanism to control information flow by D. Denning [20] was based on tainting methods : when modifying a variable, the security level of that variable becomes the highest security level of the variables used to produce the value to store in the variable.

The thesis of G. Le Guernic [29] proposes and proves a precise dynamic analysis for non-interference. The monitor keeps up to date the set of *entities* (variables, program counter, ...) that may be different in other executions and authorize, denies or edits the outputs depending on the state of the monitor.

T. Austin and C. Flanagan also propose sound dynamic analyses for non-interference based on the *no-sensitive-upgrade* policy [5] and the *permissive upgrade* policy [7].

In 2010, D. Devriese and F. Piessens [22] introduced the notion of *secure multi-execution* allowing a sound and precise technique for information flow verification by executing a program multiple times with different security levels. Inspired by this work, T. Austin and C. Flanagan [6] present a new dynamic analysis for information flow based on *faceted values* : they contain two values to be used in different situations, one for each security level of the current execution. 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.

Our approach is similar to these works in the sense that it is based on actual executions, but we consider every execution whereas they monitor a single execution, modifying it if it is interferent. Our goals are also quite different : they provide a monitor, we provide a framework to simplify the certification of analyses.

Static Analyses

A. Sabelfeld and A. Russo [34, 36] 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 [9] 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 [27] present a family of semantically sound type system for non-interference. The main relation between their paper and this work is the use of dependencies : a mapping from a variable to sets of variables they depend on in [27], 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. Once again, we do not propose an analysis, but a generic way to mechanically build the refined semantics.

Different types of non-interference

There are several modern definitions of non-interference. In particular, non-interference may take into account the termination of an execution of the program (*termination-insensitive non-interference* [3], *termination-aware non-interference* [12]), or the time elapsed during the execution of the program (*timing- and termination-sensitive non-interference* [28]). Our work only considers termination-insensitive non-interference, but to be able to deal with non-terminating executions, we speculate we would only need to

consider a co-inductive version of the semantics we give here. We did not go further into this conjecture and all the theorems would require different formal proofs to fit the termination-sensitive definition of the non-interference property.

Other hyperproperties E. Cecchetti, A.C. Myers and O. Arden [15] introduced *nonmalleable information flow*, a property generalizing non-interference. They show it is a *4-safety hyperproperty*, the first formation security property based on more than two executions. We believe the annotations in our work may be adapted to fit to this hyperproperty.

1.2 Coq proof assistant

Every formal proof has been conducted thanks to the coq proof assistant. We briefly describe what Coq is and present some features used in our work.

1.2.1 A proof assistant

Coq is a tool providing a language to formalize mathematical problems, state properties and then prove them semi-automatically.

The development of Coq started in 1984 and is based on the Calculus Of Construction by T. Coquand and G. Huet [19]. Among many theorems and properties, it is notable that Coq has been used to prove the 4-color theorem [24], the Feit-Thompson theorem [25] (or odd order theorem) and the certified C compiler CompCert [30].

Coq allows to formalize mathematics with type theory thanks to the *Curry-Howard* correspondence relating types with properties and programs of a type with proofs of the corresponding theorem. Then it is possible to prove property about that formalization thanks to instructions called *tactics*. These instructions, if correctly executed, will build a proof of the desired property.

For example let us try to prove the following commutativity property of the *or* operator over the formulas of the propositional calculus. First of all we must state the property :

```
Lemma or_comm : forall A B : Prop, A \/ B -> B \/ A.
```

We can start the proof mechanism by the keyword *Proof* :

```
Proof.
```

This instruction triggers a goal (initially the property to prove) and and environment of hypotheses. We begin by introducing 2 proposition A and B in the environment and removing them from the goal.

```
  intros A B.
```

Then we can introduce the hypothesis and call it HAB.

```
  intros HAB.
```

This hypothesis gives us the information that either A is true or B is true. The tactic *destruct* allows to destruct the hypothesis in two parts (HA and HB) and also to duplicate the goals. In both goals, HAB is replaced by either HA or HB.

```
  destruct HAB as [HA HB].
```

We now have to prove `B \/ A` in an environment where A is true and in another environment where B is true. We use the symbol $+$ to focus on the first subgoal and once it will be proved, we will be able to focus on the other one.

```
  + right.
    assumption.
```

In the first case, the tactic *right* explicitly ask to prove only the right part of the goal when the goal is a disjunction and *assumption* declares that the new goal is exactly one of our hypotheses, i.e. A in that case.

```
  + left.
    assumption.
```

Finally, the second proof is taken care of in a similar way.

Another important feature of the Coq language that we used in the developments of this thesis is the type class mechanism.

1.2.2 Type classes

Type classes were first introduced in Haskell in 1979 [39] and adapted to Coq in 2008 by M. Sozeau and N. Oury [37]. The goal of type classes is to be able to program with a abstract description of a type. It perfectly fits our case since we want to mechanically derive a multisemantics for any language given in PBS form.

As an example let us define monoids in coq. A monoid is a mathematical structure made of a set of elements, an associative binary operator and an identity element.

```
Class Monoid : Type := {
  A: Type;
  dot : A -> A -> A;
  e : A;
}.
```

This definition does not perfectly fit the monoid definition since we only gave the fields of our class. In coq, since properties are types, we can add proofs in the fields. This way we can specify the associativity of dot and the identity property of one.

```
Class Monoid : Type := {

  A: Type;
  dot : A -> A -> A;
  e : A;

  dot_associative : forall x y z : A,
  dot (dot x y) z = dot x (dot y z);
  identity_left : forall x : A, dot e x = x;
  identity_right : forall x : A, dot x e = x;

}.
```

This definition of monoids allows to state properties over monoids and prove them. For instance, let us prove that the neutral element is idempotent.

```
Lemma e_idempotent :
dot e e = e
.
Proof.
refine (identity_left e).
Qed.
```

Later, we may want to concretize the type class to use our lemma on a particular monoid. In our case we can instantiate our monoids with natural numbers :

```
Require Import NArith.

Instance myZ : Monoid := {
  A := nat;
  dot := plus;
  e := 0;

  dot_associative := PeanoNat.Nat.add_assoc;
  identity_left := PeanoNat.Nat.add_0_l;
  identity_right := PeanoNat.Nat.add_0_r;
}.
```

To have access to the proofs of associativity and identity in *nat*, we import the *NArith* module. We can then fill each field of the monoid with the corresponding elements in the particular case of the natural numbers.

Now we can use the proven lemma over all the monoids to prove the following lemma :

```
Lemma plus00 : 0+0=0.
Proof.
  refine (e_idempotent myNat).
Qed.
```

Another way to verify the concretization of this lemma is simply checking if the type of the instantiated lemma is the expected one :

```
Check (e_idempotent myNat : 0+0=0).
```

1.2.3 Partial functions

For clarity reasons, and in this section only, we emphasis the difference between *mathematical objects* and their `encodings` by using different fonts.

There are two main ways to formally encode partial functions. The encoding restricts either the domain or the image of the function. In the first case, one could define a new type to represent the domain on which the function is mathematically defined and encode the Coq function on this new type. In our work, we have a lot of partial functions and their domains are various therefore parameterizing a function by a type representing its domain is uselessly complex. Instead, we chose to represent them by restricting the image of the functions with the option type.

For instance a mathematical partial function

$$f : E \to F$$

is encoded by a function

```
f : E -> option F
```

such that, for all elements $x \in E, y \in F$, if $f(x) = y$ then `f(x)` returns `Some y`; and for all element x out of the domain of f, `f(x)` returns `None`.

1.3 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 [16] is not only concise, it has been shown to scale to complex programming languages while still being amenable to formalization with a proof assistant [13]. We slightly modify the definition of Pretty-Big-Step to make it more uniform and to simplify the definition of non-interference.

1.3.1 What is PBS ?

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 term under evaluation and the current state to decide which rule applies. To illustrate the Pretty-Big-Step approach, let us consider the evaluation of a conditional. It may look like these two rules in Big-Step format.

$$\textsc{IfTrue}\ \frac{M, e \to (M', true) \qquad M', s_1 \to M''}{M, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to M''}$$

$$\textsc{IfFalse}\ \frac{M, e \to (M', false) \qquad M', s_2 \to M''}{M, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to M''}$$

where e, s_1 and s_2 are terms, and M, M' and M'' are memory state.

Although these rules only have two inductive premises each, one has to partially execute them to know which one is applicable. In Pretty-Big-Step, one first evaluates the expression 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 the state in which they are evaluated often include previously computed values. Here are the rules for evaluating a conditional in Pretty-Big-Step.

$$\textsc{If}\ \frac{M, e \to (M', v) \qquad (M', v), \texttt{If}_1\ s_1\ s_2 \to M''}{M, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to M''}$$

$$\textsc{IfTrue}\ \frac{M, s_1 \to M'}{(M, true), \texttt{If}_1\ s_1\ s_2 \to M'} \qquad \textsc{IfFalse}\ \frac{M, s_2 \to M'}{(M, false), \texttt{If}_1\ s_1\ s_2 \to M'}$$

The evaluation of the expression has been factorized into one single rule IF.

Formally, rules are in three groups :

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.

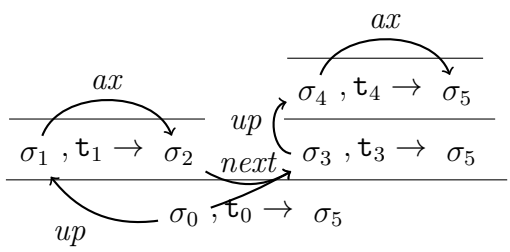
$$Ax\ \frac{}{\sigma, t \to \sigma'} \qquad R_1\ \frac{\sigma_1, t_1 \to \sigma_1'}{\sigma, t \to \sigma'} \qquad R_2\ \frac{\sigma_1, t_1 \to \sigma_1' \qquad \sigma_2, t_2 \to \sigma_2'}{\sigma, t \to \sigma'}$$

FIGURE 1.7 – Types of rules for a Pretty-Big-Step semantics

In Pretty-Big-Step, rules may take as input a memory and zero, one, or several values and they may either return a memory or a memory and also several value. To account for this in a uniform way, we define a state $\sigma \in State$ as a pair of a memory and a list of values, called an *extra*. We write $extra(\sigma)$ to refer to the list of values in a state σ. As often as possible, we will denote state with the letter σ ($\sigma_1, \sigma', \ldots$) and memories with the letter M ($M_1, M', \ldots$). To simplify notations, if the extra is an empty list, we omit the extra and we only write the memory ; and if the extra is a singleton then we denote the state as a pair of a memory and the value in the extra.

A rule is entirely defined by the following components.

- Axioms
  - $t : term$, the term on which the axiom can be applied ;
  - $ax : State \to State$, a function that give the resulting state given the initial state.
- Rule 1
  - $t : term$, the term on which the rule 1 can be applied ;
  - $t_1 : term$, a term to evaluate in order to continue the derivation ;
  - $up : State \to State$, a function that returns the new state in which t_1 will be evaluated.
- Rule 2
  - $t : term$, the term on which the rule 2 can be applied ;
  - $t_1, t_2 : term$, the terms to derive in order to get the result for t ;
  - $up : State \to State$, a function returning the state in which the term t_1 has to be derived ;
  - $next : State * State \to State$, a function giving the state in which t_2 had to be derived depending on the initial state and the result of the derivation of t_1.



The functions ax, up, and $next$ are partial functions because the rules may not be defined for every state. For example, the rule IFTRUE above is defined only when the state has a single extra that is the boolean value $true$.

Figure 1.8 shows again the three types of rules but with their explicit components.

$$Ax\ \frac{}{\sigma, t \to ax(\sigma)} \qquad R_1\ \frac{up(\sigma), t_1 \to \sigma'}{\sigma, t \to \sigma'}$$

$$R_2\ \frac{up(\sigma), t_1 \to \sigma'_1 \qquad next(\sigma, \sigma'_1), t_2 \to \sigma'}{\sigma, t \to \sigma'}$$

FIGURE 1.8 – Types of rules for a Pretty-Big-Step semantics (bis)

When describing a derivation $\sigma, t \to \sigma'$ (or a rule with this derivation as conclusion), we will refer to σ as the *semantic context* of the derivation (or the rule) and to σ' as the *result* of the derivation (or the rule).

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

— If the evaluation is immediate, we can directly give the results (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 for each possible branching. This is used for instance after evaluating the condition in a conditional statement.

— If the evaluation first needs to 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. This is how the conditional

statement works : first evaluate the guard and then compute another term in a state containing the result of the evaluation.

As an example, Figure 1.9 shows the derivation of the program

$$\texttt{if}\ x\ \texttt{then}\ 0\ \texttt{else}\ 1$$

in a state for which the stored value of x is $true$.

$$\dfrac{\dfrac{}{M, x \to (M, true)} \qquad \dfrac{\dfrac{}{M, 0 \to (M, 0)}}{(M, true), \texttt{If}_1\ (0)\ (1) \to (M, 0)}}{M, \texttt{if}\ x\ \texttt{then}\ 0\ \texttt{else}\ 1 \to (M, 0)}$$

FIGURE 1.9 – Derivation of a simple program

1.3.2 Advantages

Modularity

The first advantage of Pretty-Big-Step is that it is very modular : extending the language can be done by adding new rules and without modifying previous ones. For instance, let us suppose we have a C-like *for loop* :

`for`(initialization, condition, step){body}

and see the differences in Big-Step and Pretty-Big-Step when adding the notion of errors. As shown in Figure 1.10, in Big-Step style, there are two rules to write to handle the for loop : one for each evaluation of the condition. In the first case, we start the loop by initializing, then we test for the condition, if the result is *true* we continue by evaluating the body, we proceed by doing the step for the next loop and finally we go back to the beginning of the loop ignoring the initialization phase. The second case also starts with the initialization and the evaluation of the condition, but if the evaluation returns *false*, the execution stops here.

$$\textsc{ForTrue}\ \frac{M_0, init \to M_1 \qquad M_1, cond \to (M_2, true) \qquad M_2, body \to M_3 \qquad M_3, step \to M_4 \qquad M_4, \texttt{for}(skip, cond, step)\{body\} \to M_5}{M_0, \texttt{for}(init, cond, step)\{body\} \to M_5}$$

$$\textsc{ForFalse}\ \frac{M_0, init \to M_1 \qquad M_1, cond \to (M_2, false)}{M_0, \texttt{for}(init, cond, step)\{body\} \to M_2}$$

FIGURE 1.10 – Rules of the for loop in Big-Step

Conversely, Figure 1.11 shows the 6 rules needed to handle the for loop in Pretty-Big-Step style. They introduce intermediate terms $\texttt{for}_1$, $\texttt{for}_2$ and $\texttt{for}_3$. They do not have the parameter $init$ since these new terms corresponds to the loop phases.

$$\textsc{For}\ \frac{M, init \to \sigma \qquad \sigma, \texttt{for}_1(cond, step)\{body\} \to \sigma'}{M, \texttt{for}(init, cond, step)\{body\} \to \sigma'}$$

$$\textsc{For1}\ \frac{M, cond \to \sigma \qquad \sigma, \texttt{for}_2(cond, step)\{body\} \to \sigma'}{M, \texttt{for}_1(cond, step)\{body\} \to \sigma'}$$

$$\textsc{For2True}\ \frac{M, body \to \sigma \qquad \sigma, \texttt{for}_3(cond, step)\{body\} \to \sigma'}{(M, true), \texttt{for}_2(cond, step)\{body\} \to \sigma'}$$

$$\textsc{For2False}\ \frac{}{(M, false), \texttt{for}_2(cond, step)\{body\} \to M}$$

$$\textsc{For3}\ \frac{M, step \to \sigma \qquad \sigma, \texttt{for}_1(cond, step)\{body\} \to \sigma'}{M, \texttt{for}_3(cond, step)\{body\} \to \sigma'}$$

FIGURE 1.11 – Rules of the for loop in Pretty-Big-Step

To derive the rule FOR, one needs to first evaluate the initialization $init$ and then $\texttt{for}_1$ as the continuation. This first extended term is evaluated with the rule FOR1 which evaluates the condition and then lets the term $\texttt{for}_2$ decide to take the loop or not depending on the value calculated. The term $\texttt{for}_2$ has then 2 ways to be evaluated : either the extra is $true$ and in that case the rule FOR2TRUE applies or it is $false$ and then the rule FOR2FALSE

applies. In the first case, the program enters in the loop evaluating $body$ and letting $\texttt{for}_3$ taking care of ending the loop. In the second case there is no computation left and the loop ends here. Once the body is executed, the rule FOR3 proceeds to evaluate the $step$ statement and continues with the same continuation as after the initialization : $\texttt{for}_1$.

It is important to note that, despite the increasing number of rules, the number of inductive premises stays approximately the same : 7 premises in Big-Step and 8 in Pretty-Big-Step.

Let us extend the language with an error exception and commands to throw and catch this error and compare the changes to operate in Big-Step and in Pretty-Big-Step.

Error handling in Big-Step

In the Big-Step version, we suppose we have the rules of Figure 1.12 to throw and catch errors.

$$\textsc{Throw}\ \frac{}{M, \texttt{throw} \rightarrow (M, err)}$$

$$\textsc{TryCatch}\ \frac{M, body \rightarrow (M', err) \qquad M', s_{error} \rightarrow \sigma'}{M, \texttt{try}\ body\ \texttt{catch}\ s_{error} \rightarrow \sigma'}$$

$$\textsc{TryNoCatch}\ \frac{M, body \rightarrow \sigma' \qquad \sigma' \neq (M', err)}{M, \texttt{try}\ body\ \texttt{catch}\ s_{error} \rightarrow \sigma'}$$

FIGURE 1.12 – Rules of throw and catch Big-Step style

To handle the interaction of errors with loops, one needs to add the 5 rules of Figure 1.13. Each rule corresponds to a step that could throw an error during the evaluation : the initialization, the condition, the body, the step and the next loops. We added 5 rules with a total of 15 premises.

Error handling in Pretty-Big-Step

In Pretty-Big-Step, it is way simpler because the structure allows to propagate the errors between each intermediate step thanks to the extra. First let us give the rules to throw and catch errors in Figure 1.14

$$\textsc{ForErr1}\ \frac{M_0, init \to (M_1, err)}{M_0, \texttt{for}(init, cond, step)\{body\} \to (M_1, err)}$$

$$\textsc{ForErr2}\ \frac{M_0, init \to M_1 \qquad M_1, cond \to (M_2, err)}{M_0, \texttt{for}(init, cond, step)\{body\} \to (M_2, err)}$$

$$\textsc{ForErr3}\ \frac{\begin{array}{c} M_0, init \to M_1 \\ M_1, cond \to (M_2, true) \qquad M_2, body \to (M_3, err) \end{array}}{M_0, \texttt{for}(init, cond, step)\{body\} \to (M_3, err)}$$

$$\textsc{ForErr4}\ \frac{\begin{array}{c} M_0, init \to M_1 \qquad M_1, cond \to (M_2, true) \\ M_2, body \to M_3 \qquad M_3, step \to (M_4, err) \end{array}}{M_0, \texttt{for}(init, cond, step)\{body\} \to (M_4, err)}$$

$$\textsc{ForErr5}\ \frac{\begin{array}{c} M_0, init \to M_1 \\ M_1, cond \to (M_2, true) \qquad M_2, body \to M_3 \qquad M_3, step \to M_4 \\ M_4, \texttt{for}(skip, cond, step)\{body\} \to (M_5, err) \end{array}}{M_0, \texttt{for}(init, cond, step)\{body\} \to (M_5, err)}$$

FIGURE 1.13 – Extra rules about for/catch interaction (BS)

$$\textsc{Throw}\ \frac{}{M, \texttt{throw} \to (M, err)} \qquad \textsc{Try}\ \frac{M, body \to \sigma \qquad \sigma, \texttt{catch}\ s_{error} \to \sigma'}{M, \texttt{try}\ body\ \texttt{catch}\ s_{error} \to \sigma'}$$

$$\textsc{Catch}\ \frac{M, s_{error} \to \sigma'}{(M, err), \texttt{catch}\ s_{error} \to \sigma'}$$

$$\textsc{NoCatch}\ \frac{l_{extra} \neq [err]}{(M, l_{extra}), \texttt{catch}\ s_{error} \to M}$$

FIGURE 1.14 – Rules of throw and catch Pretty-Big-Step style

Now to handle the interactions of errors with other terms, we need to add the rule of Figure 1.15 for every rule 2 over a term t and with t_1 as term for the first subderivation, excepted for the rule TRY.

In any rule 2, when an error occurs in the left branch, the rule ERR automatically propagates the error without having to evaluate the continuation. If

$$\textsc{Err}\ \frac{up(\sigma), t_1 \rightarrow (M, err)}{\sigma, t \rightarrow (M, err)}$$

FIGURE 1.15 – Extra rules about for/catch interaction (PBS)

an error appears in the subderivation of a rule 1, the error is naturally transmitted to the result. In the case of the loop it corresponds to a total of 4 additional rules and only 4 premises.

On one hand, the adding of the new rules in Pretty-Big-Step is way simpler since it matches the total number of rules 2 whereas in Big-Step many rules need to be added to manage errors in a rule with more than two premises. On the other hand, the additional rules in PBS are fewer and smaller in terms of premises because there is no redundant subderivation in different rules.

Abstraction

The second advantage is that Pretty-Big-Step is really easy to abstract. Our formalization is stricter than Charguéraud initial version of Pretty-Big-Step. The extra part, which Charguéraud included in the subterm to evaluate, is now in the state. It allows to fully define each rule by a strict scheme depending on which kind of rule it is (axiom, Rule1 or Rule2). We will see in Chapter 2 that we can work with a Pretty-Big-Step language without concretizing the terms of the language, only considering the structure of the rules. This is an important property since it allow any language to fit in our work at the only condition that it is written in Pretty-Big-Step form.

1.3.3 A While language in PBS

To illustrate our approach, we introduce a small WHILE language suitable for non-interference. It is a classical WHILE language with input/output commands to receive and send data. We first give the syntax of the language and then its semantics in Pretty-Big-Step form.

Memory model We propose to model non-interference by making explicit the inputs of a program and its outputs. We do not consider interactive pro-

grams, 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, a set of variables Var, a set of inputs $Inputs$ and a set of outputs $Outputs$. 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.

$$Env_i := Inputs \mapsto Val$$
$$Env_x := Var \mapsto Val$$
$$Env_o := Outputs \mapsto List(Val)$$
$$Mem := Env_i \times Env_x \times Env_o$$
$$Extra := List(Val)$$
$$State := Mem \times Extra$$

For the purpose of notation, when there is no ambiguity, memories and states may be seen has functions from $Inputs$, Var, or $Outputs$ to Val or $List(Val)$ to represent the part of the memory that should be used. For example, the value stored in the variable x in a memory M or a state σ may be written $M(x)$ or $\sigma(x)$.

Syntax In this language, we distinguish expressions and statements but they formally both are defined as terms. An expression is either a constant value, a variable, an input, or the addition of two expressions. A statement is either a no-op operation `skip`, a sequence of two statements, a conditional, a while loop, an assignment of an expression into a variable, or an assignment of an expression into an output.

⟨*term*⟩ `t` ::= `Const n` | `Var x` | `Input i` | `Plus t t` | `Skip` | `Seq t t` | `If t t t` | `While t t` | `Assign x t` | `Output o t`

We add to the language the extended terms required by the Pretty-Big-Step format.

$\langle$*term*$\rangle$ `t ::= ... | Plus1 t | Plus2 | Seq1 t | If1 t t | While1 t t | While2 t t`
`| Assign1 x | Output1 o`

Semantics The difference between expressions and statements is at the semantic level where expressions always return a value, while statements may do so (the if statement for example) but it is not always the case.

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

c for `Const` c
x for `Var` x
$e_1 + e_2$ for `Plus1` e_2
$+_1\ e_2$ for `Plus2`
$+_2$ for `Plus` $e_1\ e_2$
$s_1; s_2$ for `Seq` $s_1\ s_2$
$;_1\ s_2$ for `Seq1` s_2
$x := e$ for `Assign` $x\ e$
$x :=_1$ for `Assign1` x
`if` e `then` s_1 `else` s_2 for `If` $e\ s_1\ s_2$
`If`$_1$ $s_1\ s_2$ for `If1` $s_1\ s_2$
`while` e `do` s for `While` $e\ s$
`while`$_1$ e `do` s for `While1` $e\ s$
`while`$_2$ e `do` s for `While2` $e\ s$

To evaluate a constant c the axiom CST requires the semantic context to be a memory and an empty extra, and returns a result formed by the same memory and an extra containing the value c. The rule VAR looks up for the value stored in x and returns a state made of the memory unmodified and the value found. The rule INPUT works exactly the same way but in the input environment.

The addition $e_1 + e_2$ of two expressions is managed by three rules PLUS, PLUS1 and PLUS2. The first one is a rule 2 in which the first premise derives the first expression e_1 and the result of this derivation becomes the semantic context of the second premise. This second premise is the derivation of $+_1\ e_2$ and requires the rule PLUS1. This rule is also a rule 2 : its first premise is the derivation of e_2 and its second premise is the derivation of $+_2$. The extra

of the current rule (corresponding to the value of the first expression e_1) is added to the extra in the result of the derivation of e_2 to form the semantic context of the continuation. Finally, the axiom PLUS2 can add the two values currently in the extra to produce the final sum.

To derive the `skip`, the rule SKIP needs a state with an empty extra and returns the same state.

The sequence is derived with the rules SEQ and SEQ1. The first rule is a rule 2 that derives the first statement and passes the result to the continuation $;_1$. Then the second rule is a rule 1 simply deriving the second statement.

In the rule IF, the derivation of `if` e `then` s_1 `else` s_2 starts with the derivation of the guarding condition e. The result is passed to the extended statement $\texttt{If}_1\ s_1\ s_2$. We then have two rules to evaluate $\texttt{If}_1\ s_1\ s_2$, one for each possible extra. The first one, IFTRUE, is a rule 1 in which the premise is the derivation of the first branch granted that the value in the extra is $true$. The second one, IFFALSE, derives the second branch when the extra is $false$.

The rules to derive `while` e `do` s are similiar to those to derive an if statement. First the rule WHILE derives the guarding condition. Then two cases can appear. Either we have to derive $\texttt{while}_1\ e\ \texttt{do}\ s$ in a state with an extra containing $false$ and in that case the axiom WHILEFALSE simply returns the state without the extra. Or we have to derive the extended term in a state with $true$ in the extra and then firstly the rule WHILETRUE1 derives the body of the loop and lets the continuation $\texttt{while}_2\ e\ \texttt{do}\ s$ decide the remaining derivations to do, secondly the rule WHILETRUE2 branches back to the beginning of the loop deriving again `while` e `do` s.

An assignment $x := e$ is derived by the rule 2 ASG that derives the expression e and gives the result to the continuation $x :=_1$. This extended term can be derived by the rule ASG1 in an state containing a value v in the extra. The resulting state is the same than the semantic context with v stored in x instead of the previous value.

The output `Ouput` o e of an expression e to an output o is similar : first the rule OUTPUT derives the expression to get its value and gives it to the continuation, then the continuation is derived with the rule OUTPUT1 which adds the value of the extra to the list of values already output by o.

$$\textsc{Cst}\ \frac{}{M, c \to (M, c)} \qquad \textsc{Var}\ \frac{M(x) = v}{M, x \to (M, v)} \qquad \textsc{Input}\ \frac{M(i) = v}{M, \texttt{Input}\ i \to (M, v)}$$

$$\textsc{Plus}\ \frac{M, e_1 \to (M', v_1) \qquad (M', v_1), +_1\ e_2 \to (M'', v)}{M, e_1 + e_2 \to (M'', v)}$$

$$\textsc{Plus1}\ \frac{M, e_2 \to (M', v_2) \qquad (M', [v_1, v_2]), +_2 \to M'', v}{(M, v_1), +_1\ e_2 \to M'', v}$$

$$\textsc{Plus2}\ \frac{v = v_1 + v_2}{(M, [v1, v2])), +_2 \to (M, v)} \qquad \textsc{Skip}\ \frac{}{M, \texttt{skip} \to M}$$

$$\textsc{Seq}\ \frac{M, s_1 \to M' \qquad M', ;_1\ s_2 \to M''}{M, s_1; s_2 \to M''} \qquad \textsc{Seq1}\ \frac{M, s \to M'}{M, ;_1\ s \to M'}$$

$$\textsc{If}\ \frac{M, e \to (M', v) \qquad (M', v), \texttt{If}_1\ s_1\ s_2 \to M''}{M, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to M''}$$

$$\textsc{IfTrue}\ \frac{M, s_1 \to M'}{(M, true), \texttt{If}_1\ s_1\ s_2 \to M'} \qquad \textsc{IfFalse}\ \frac{M, s_2 \to M'}{(M, false), \texttt{If}_1\ s_1\ s_2 \to M'}$$

FIGURE 1.16 – Rules of WHILE (1)

$$\textsc{While}\ \frac{M, e \to (M', v) \qquad (M', v), \texttt{while}_1\ e\ \texttt{do}\ s \to M''}{M, \texttt{while}\ e\ \texttt{do}\ s \to M''}$$

$$\textsc{WhileFalse}\ \frac{}{(M, false), \texttt{while}_1\ e\ \texttt{do}\ s \to M}$$

$$\textsc{WhileTrue1}\ \frac{M, s \to M' \qquad M', \texttt{while}_2\ e\ \texttt{do}\ s \to M''}{(M, true), \texttt{while}_1\ e\ \texttt{do}\ s \to M''}$$

$$\textsc{WhileTrue2}\ \frac{M, \texttt{while}\ e\ \texttt{do}\ s \to M'}{M, \texttt{while}_2\ e\ \texttt{do}\ s \to M'}$$

$$\textsc{Asg}\ \frac{M, e \to (M', v) \qquad (M', v), x :=_1 \to M''}{M, x := e \to M''} \qquad \textsc{Asg1}\ \frac{M' = M[x \mapsto v]}{(M, v), x :=_1 \to M'}$$

$$\textsc{Output}\ \frac{M, e \to (M', v) \qquad (M', v), \texttt{Ouput}_1\ o \to M''}{M, \texttt{Ouput}\ o\ e \to M''}$$

$$\textsc{Output1}\ \frac{M' = M[o \mapsto v :: M(o)]}{(M, v), \texttt{Ouput}_1\ o \to M'}$$

FIGURE 1.17 – Rules of WHILE (2)

CHAPITRE 2

GENERIC FORMAT OF SEMANTICS

2.1 Motivation

As we stated before, we want our framework to work on a large variety of languages. The only constraints we enforce is that the languages semantics must be written in PBS form. It allows our framework to be independent of any language.

In order to be able to fit any PBS semantic in this work, we developed a formal PBS structure in which the three types of rules are formally defined. It gives the possibility to think of any language only in terms of axioms, rules 1 and rules 2 ; and to totally abstract the proofs from any particular language.

One other main reason to this choice is that JavaScript is widely used in browser and web application (which are uses for which non-interference makes sense to study) and already has a PBS semantics called *JScert* [13]. This semantics is a huge inductive definition with more than 800 rules and hinders formal proofs as Coq runs out of memory when performing an inversion or an induction.

Moreover, given a PBS semantic of any language in this formalism, we want to automatically derive the associated multisemantics that we will build in section 3.

2.2 Formal Pretty-Big-Step

2.2.1 Abstract PBS

We first define what a syntax is. It simply consists of terms and values. Such a syntax can be assumed by the use of type classes.

```
Class AbstractSyntax := {
                    Term : Type;
                    Value : Type
                        }.
Context {Syntax : AbstractSyntax}.
```

Now that we have some terms and values, we can define variables as strings and then describe our memory model as in subsection 1.3.3. A State is made of 4 parts : a memory for the inputs, one for the variables, a last one for the outputs, and the extra.

```
Definition variable : Type := string.
Definition input : Type := string.
Definition output : Type := string.

Record State :=
  mkState
    {
      Envi : input -> Value;              (* input environment *)
      Envx : variable -> option value;    (* variable environment *)
      Envo : output -> list value;        (* output environment *)
      extra : list value                  (* extra *)
    }
.
```

Once the memory model is defined, we have the ability to formally define the PBS rules. There are three kinds of rules :

1. To entirely define **an axiom**, we exactly need the term t on which it applies and the function ax returning the resulting state ; for the purpose of capturing non-interference by the multisemantics we also add four sets and a boolean value : 2 sets for the inputs and variables read by the axiom, 2 sets for the variables and outputs written by the axiom and a boolean value to specify if the axiom produces an extra or if it returns an empty one. These parameters are not mandatory for a PBS but they will be used and explained in a more detailed way in chapter 4.

2. A **Rule 1** is defined by the term t on which it applies, the term t_1 needed to derive, and the function up returning the semantic context in which t_1 must be derived.
3. And a **Rule 2** is defined by the term t on which it applies, the terms t_1 and t_2 that need to be derived, the function up returning the semantic context in which t_1 must be derived and the function $next$ returning the semantic context in which t_2 must be derived.

```
Inductive rule :=
| Ax : Term                          (*t*)
       -> (State -> option State)    (*ax*)
       -> fset input                 (*Inputs read*)
       -> fset variable              (*Variables read*)
       -> fset variable              (*Variables written*)
       -> fset output                (*Outputs written*)
       -> bool                       (*extra produced*)
       -> rule
| R1 : Term                          (*t*)
       -> Term                       (*t1*)
       -> (State -> option State)    (*up*)
       -> rule
| R2 : Term                                    (*t*)
       -> Term                                 (*t1*)
       -> Term                                 (*t2*)
       -> (State -> option State)              (*up*)
       -> (State -> State -> option State)     (*next*)
       -> rule
.
```

Now that we gave a structure to our rule format we can formally define a semantics, i.e. a function giving, for each term, a list of rules that can be applied to the term. As for syntax, let us assume we have a semantics.

```
Class AbstractSemantics := {
                              Rules : Term -> list rule
```

```
                                }.
Context {Semantics : AbstractSemantics}.
```

The definition of a derivation is now possible. A derivation of term t from the semantic context σ to the result out is defined inductively with these 3 cases :

```
Inductive deriv (t:Term) (sigma:State) (out:State) : Prop :=
```

1. if there is axiom R in the semantics such that $ax(\sigma) = Some\ out$

```
  | deriv_Ax R ax ri rx wx wo pe
             (eqR : R = Ax t ax ri rx wx wo pe)
             (isRule : List.In R (Rules t))
             (eqAx : ax sigma = Some out)
    : (* ======== *)
      deriv t sigma out
```

2. if there is a rule 1 R in the semantics and a state σ_1, such that $up(\sigma) = Some\ \sigma_1$ and there is a derivation of t_1 from σ_1 to out.

```
  | deriv_R1 R t1 up sigma1
             (eqR : R = R1 t t1 up)
             (isRule : List.In R (Rules t))
             (eqUp : up sigma = Some sigma1)
             (STEP : deriv t1 sigma1 out)
    : (* ======== *)
      deriv t sigma out
```

3. if there is a rule 2 R in the semantics and three states σ', out' and σ'' such that $up(\sigma) = Some\ \sigma'$, there is a derivation of t_1 from σ' to out', $next(\sigma, out') = Some\ \sigma''$ and there is a derivation of t_2 from σ'' to out.

```
  | deriv_R2 R t1 t2 up next sigma' out' sigma''
             (eqR : R = R2 t t1 t2 up next)
             (isRule : List.In R (Rules t))
             (eqUp : up sigma = Some sigma')
             (STEP1 : deriv t1 sigma' out')
             (eqNext : next sigma out' = Some sigma'')
```

```
                (STEP2 : deriv t2 sigma'' out)
        : (* ======== *)
          deriv t sigma out
      .
```

2.2.2 Concretization of the WHILE language

To illustrate a concretization of a pbs language, we formalize the while language described in subsection 1.3.3. Values and Terms are defined inductively. Values are either integer or boolean.

```
Inductive Concr_value : Type :=
| Num (z:Z)
| Bool (b:bool)
.

Inductive Concr_Term: Type :=
(*Statements*)
| Skip : Concr_Term
| Seq : Concr_Term -> Concr_Term -> Concr_Term
| Seq1 : Concr_Term -> Concr_Term
| If : Concr_Term -> Concr_Term -> Concr_Term -> Concr_Term
| If1 : Concr_Term -> Concr_Term -> Concr_Term
| While : Concr_Term -> Concr_Term -> Concr_Term
| While1 : Concr_Term -> Concr_Term -> Concr_Term
| While2 : Concr_Term -> Concr_Term -> Concr_Term
| Assign : variable -> Concr_Term -> Concr_Term
| Assign1 : variable -> Concr_Term
| Out : output -> Concr_Term -> Concr_Term
| Out1 : output -> Concr_Term

(*Expressions*)
| Var : variable -> Concr_Term
| Cons : Concr_value -> Concr_Term
| Plus : Concr_Term -> Concr_Term -> Concr_Term
```

```
| Plus1 : Concr_Term  -> Concr_Term
| Plus2 : Concr_Term
| In : input -> Concr_Term
.
```

These values and terms form our syntax :

```
Instance Concr_Syntax : AbstractSyntax:=
{
  Value := Concr_value;
  Term := Concr_Term
}.
```

We only describe here the rules for the Skip, If, Out, Var, Plus and In terms. The complete Coq code can be found in Appendix A. The appendix also shows 3 functions to change the extra of a state (`update_extra`), to modify the value of a variable (`update_var`) or an output (`update_output`). The rules are given by a function returning, for each term, a list of rules that can be applied to it.

```
Definition Concr_Rules (t:Concr_Term): list rule :=
  match t with
```

The only rule for the term Skip is an axiom : the term it applies on is of course Skip. The ax function is a partial function defined only if the extra is empty : if it is empty, ax returns the semantic context as the result and if it is not ax returns `None` to illustrate that there is no possible derivation. All of the sets are empty since the rule does not read or write any input, variable or output. Additionally, this rule does not produce an extra therefore the boolean value is $false$.

```
| Skip => (Ax Skip
              (fun sc => match extra sc with
                           | nil => Some sc
                           | _ => None end)
              empty empty empty empty
              false)
            :: nil
```

The term `If e s_1 s_2` cannot be directly derived with an axiom and needs other derivations. The rule for the if statement is a rule 2 in which the first term to derive is the guard e and the second one is the extended term $if_1\ s_1\ s_2$. The partial function up is defined only if the extra of the semantic context is empty and in that case, it returns the same semantic context for the guard. On the other hand, $next$ is defined only if up is defined and if the extra of the first derivation's result is a singleton containing a boolean value. In that case $next$ returns the result of the first derivation as semantic context for the If_1 statement. Note that this mechanism allows the presence of side effects to the memory during the derivation of the guard.

```
| If e s1 s2 => (R2 (If e s1 s2)
                    (e)
                    (If1 s1 s2)
                    (fun sc => match extra sc with
                                 | nil => Some sc
                                 | _ => None end)
                    (fun sc res => match extra sc, extra res with
                                     | nil, (Bool _) :: nil => Some res
                                     | _,_ => None end))
               :: nil
```

Once the guard has been evaluated and stored in the extra, the term $if_1\ s_1\ s_2$ can be derived in two possible ways depending on the value stored. Therefore, there are two rules for this term. Both rules are rule 1 and they can only be applied when the extra of the semantic context contains nothing more than a boolean value. If the boolean is $true$, the first rule applies and it considers s_1 for the inductive derivation. In the other case, the second rule applies and it considers s_2. Both up functions keep the memories unchanged and replace the extra by an empty list.

```
| If1 s1 s2 => (R1 (If1 s1 s2)
                   (s1)
                   (fun sc => (match extra sc with
                                 | (Bool true) :: nil =>
                                     Some (update_extra sc nil)
```

```
                                | _ => None end)))
                ::
                (R1 (If1 s1 s2)
                    (s2)
                    (fun sc => (match extra sc with
                                | (Bool false) :: nil =>
                                      Some (update_extra sc nil)
                                | _ => None end)))
                :: nil
```

To output an expression e on o, a rule 2 applies : first derive e and then $Out_1\ o$. The up partial function requires an empty extra in the semantic context and returns the same state. The $next$ function additionally requires that the extra of the first derivation's result is a singleton and returns the result of the first derivation.

```
| Out o e=> (R2 (Out o e)
                (e)
                (Out1 o)
                (fun sc => match extra sc with
                           | nil => Some sc
                           | _ => None end)
                (fun sc res => match extra sc, extra res with
                               | nil, v :: nil => Some res
                               | _,_ => None end))
            :: nil
```

Then to derive the extended term $Out_1\ o$ the rule to apply is the following axiom. The ax partial function is defined provided that the extra in the semantic context contains only one value v and it returns the same state but with v added to the output o and an empty extra. In that case, the axiom does not read any input or variable and neither it writes into a variable ; but it writes into the output o thus the fourth set is the singleton $\{o\}$. Since this axiom only sends the value in the output o, it does not produce a value in the extra. therefore the boolean value is set to $false$.

```
| Out1 o => (Ax (Out1 o)
                (fun sc => match extra sc with
                             | v::nil =>
                                   Some (update_extra
                                            (update_output sc o v)
                                             nil)
                             | _ => None end)
                empty empty empty (singleton o)
                false)
            :: nil
```

The term `Var` x also has only one rule that can be applied. It is an axiom for which the ax function requires two things on the semantic context : first, the extra should be empty and second, x should contain some value v in the variable environment (i.e. x must store something of the form $Some\ v$). If this is the case, ax returns the same states but with a list containing only the value v as extra. The rule only reads the variable x and does not write into the memory, all the sets are empty except for the set of the read variables which is $\{x\}$. This is a rule that stores a value in the extra, therefore the boolean value is set to $true$.

```
| Var x => (Ax (Var x)
               (fun sc => match extra sc, Envx sc x with
                            | nil, Some v =>
                                  Some (update_extra sc (v :: nil))
                            | _,_ => None end)
               empty (singleton x) empty empty
               true)
           :: nil
```

The rule to derive an In statement is an axiom that simply requires the semantic context to have an empty extra and the result is the same state as the semantic context with the value value stored in the i input copied in the extra. This rule obviously reads the input i and reads or writes nothing else thus the corresponding set is $\{i\}$. This axiom also stores a value in the extra, the boolean value is set to $true$.

```
| In i => (Ax (In i)
              (fun sc => match extra sc with
                           | nil =>
                              Some (update_extra sc ((Envi sc i) :: nil))
                           | _ => None
                         end)
              (singleton i) empty empty empty
              true)
          :: nil
```

This last example illustrate a case in which the extra has more than one elements in the list. The term `Plus e1 e2` can be derived by a unique rule 2 : the two expression to derive are e_1 and the extended term `Plus1 e2`. e_1 has to be derived in the same state than the initial state provided that the extra is empty. `Plus1 e2` has to be derived in the state resulting of the first derivation, provided that the extra contains only a value corresponding to a number.

```
| Plus e1 e2 => (R2 (Plus e1 e2)
                    (e1)
                    (Plus1 e2)
                    (fun sc => match extra sc with
                                 | nil => Some sc
                                 | _ => None end)
                    (fun sc res => match extra sc, extra res with
                                     | nil, (Num _)::nil => Some res
                                     | _,_ => None end))
                :: nil
```

`Plus1 e2` is also a rule 2 : the rule is similar to the previous one with some differences on the extra. The condition to derive the rule is that the extra contains a unique number n_1, which is removed when deriving e_2. But it is reintroduced in the extra of the semantic context when deriving `Plus2`, with the result n_2 of the first derivation.

```
| Plus1 e2 => (R2 (Plus1 e2)
                  (e2)
```

```
                    (Plus2)
                    (fun sc => match extra sc with
                                | (Num _)::nil =>
                                      Some (update_extra sc nil)
                                | _ => None end)
                    (fun sc res => match extra sc, extra res with
                                     | (Num n1)::nil, (Num n2)::nil =>
                                       Some (update_extra res
                                             ((Num n1)::(Num n2)::nil))
                                     | _,_ => None end))
                :: nil
```

Finally, `Plus2` is derived with an axiom : the ax function produces an unchanged state except for the extra containing the sum of the two number given in the initial extra. this axiom does not read or write in the memory at all therefore the 4 sets are empty but the boolean value is $true$ because the rule produces a value in the extra.

```
| Plus2 => (Ax (Plus2)
               (fun sc => match extra sc with
                            | (Num n1) :: (Num n2) :: nil =>
                              Some (update_extra sc
                                    (Num (n1+n2) :: nil))
                            | _ => None end)
               empty empty empty empty
               true)
           :: nil
end.
```

Once that all the rules are defined the semantics can be concretized.

```
Instance Concr_Semantics : AbstractSemantics:= {
                                                  Rules := Concr_Rules
                                                }.
```

Every theorem and property proved on the abstract PBS semantics is now instantiable by this concrete semantics.

2.3 Conclusion

We wanted to define a general way to describe a PBS semantic of a language in order to have the capacity of reasoning only in terms of rule types instead of having to reason directly on the rules. Additionally, it would be a way to automatically produce different semantics than the original one when given a particular language.

In this chapter we successfully formally defined in Coq an abstract structure for a PBS language. We also gave an example of concretization of the WHILE language we introduced in section 1.3.3. This abstract semantics will be used in chapters 3 and 4 to automatically derive respectively the *multisemantics* and the *annotated multisemantics*.

CHAPITRE 3

MULTISEMANTICS

The first step of our approach is to derive a new semantics where several derivations are considered at once. We do not simply want a set of derivations, but a *multiderivation* where applications of the same rule at the same point in the derivation are shared.

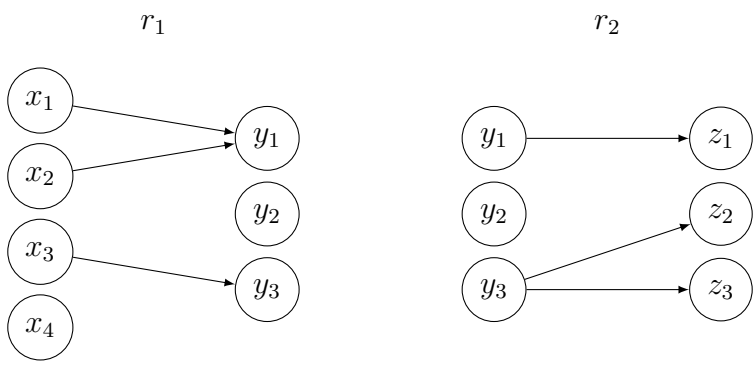
3.1 Preliminary definitions

We need a few helper functions to define the multisemantics. First, we define operators to extract the set of first and second components of a relation.

Definition 2. `fst` *and* `snd` *are defined for all relation* r *:*

$$\texttt{fst}(r) = \{x \mid (x, y) \in r\} \qquad \texttt{snd}(r) = \{y \mid (x, y) \in r\}$$

As an example, let's consider the following relations r_1 and r_2 :



In this case,

$$\begin{aligned}
&\texttt{fst}(r_1) = \{x_1, x_2, x_3\} \\
&\texttt{snd}(r_1) = \texttt{fst}(r_2) = \{y_1, y_3\} \\
&\texttt{snd}(r_2) = \{z_1, z_2, z_3\}
\end{aligned}$$

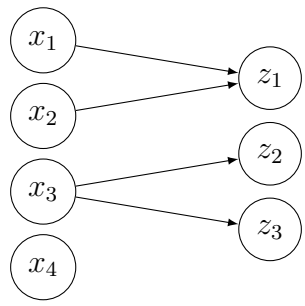
Second, we define a *strict* relation composition operator $\circ$. This operator is associative and propagates undefinedness, so we avoid using parentheses. The reason why we only allow strict compositions is that we want to forbid the composition of two non-empty relations that would result in an empty relation. An example of what could happen is given in chapter 4.

Definition 3. *For all relations r_1 and r_2 :*

$$r_1 \circ r_2 = \begin{cases} \{(x,z) | \exists y, (x,y) \in r_1 \wedge (y,z) \in r_2\} & \textit{if } \texttt{snd}(r_1) = \texttt{fst}(r_2) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

In the previous example, since $\texttt{snd}(r_1) = \texttt{fst}(r_2)$, $r_1 \circ r_2$ is defined and it is the relation

$r_1 \circ r_2$

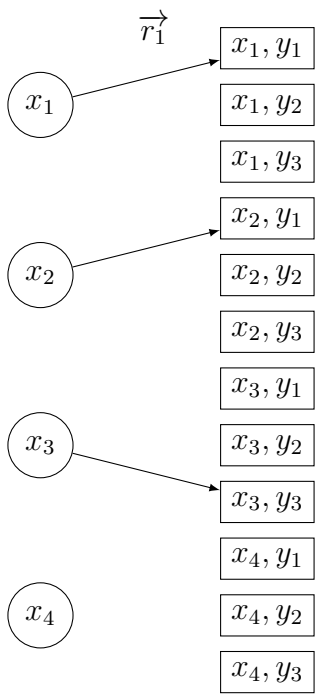


Third, we define an operator on relations $\overrightarrow{\cdot}$ that takes a relation and returns a new relation where the left-hand side is remembered in the right-hand side.

Definition 4. *For all relation r :*

$$\overrightarrow{r} = \{(\sigma, (\sigma, \sigma_1')) \mid (\sigma, \sigma_1') \in r\}$$

The following example illustrates this operator with the previous r_1 relation.



Finally, for every partial function $f : E \to F$, we define the relation $\widehat{f_{|S}} \in E \times F$ between any element of $S \subseteq E$ and its image by f if it exists.

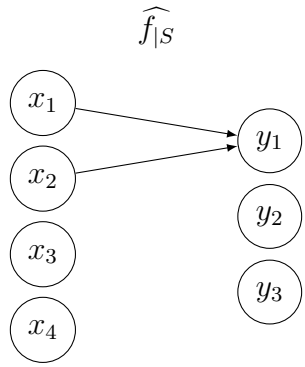
Definition 5. *For all function $f : E \to F$ and set $S \subseteq E$,*

$$\widehat{f_{|S}} = \begin{cases} \{(x, f(x)) \mid x \in S\} & \textit{if } S \subset Dom(f) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

To illustrate this definition, we consider $S = \{x_1, x_2\}$ and f the function representing the relation r_1, i.e. :

$$f(x_1) = y_1$$
$$f(x_2) = y_1$$
$$f(x_3) = y_3$$
$$x_4 \text{ has no image}$$

Since $S = \{x_1, x_2\} \subset Dom(f) = \{x_1, x_2, x_3\}$, $\widehat{f_{|S}}$ is defined and it is the relation :



3.2 Canonical structure

We use the notation $t \Downarrow \mu$ to represent a multiderivation where $\mu \subseteq State \times State$ is a relation between states. From now on, we refer to such a μ as a *multistate*. Intuitively, a multistate relates semantic contexts and results of a derivation of the considered term t. Formally, for every pair $(\sigma, \sigma') \in \mu$, we should have $\sigma, t \to \sigma'$, which is a property of the multisemantics that we state and prove in Section 3.3.

Figure 3.1 shows how to derive a rule in the multisemantics from a rule in PBS style. There are three cases as there are three kinds of PBS rules. For each case, t_1, t_2, up, and $next$ come from the corresponding PBS rule.

$$\textsc{MltAx}\ \frac{\mu = \widehat{ax_{|\mathtt{fst}(\mu)}} \qquad \mu \neq \emptyset}{t \Downarrow \mu} \qquad \textsc{MltR1}\ \frac{t_1 \Downarrow \mu_1 \qquad \mu = \widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1}{t \Downarrow \mu}$$

$$\textsc{MltR2}\ \frac{t_1 \Downarrow \mu_1 \qquad t_2 \Downarrow \mu_2 \qquad \mu' = \widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1 \qquad \mu = \overrightarrow{\mu'} \circ \widehat{next_{|\mathtt{snd}(\overrightarrow{\mu'})}} \circ \mu_2}{t \Downarrow \mu}$$

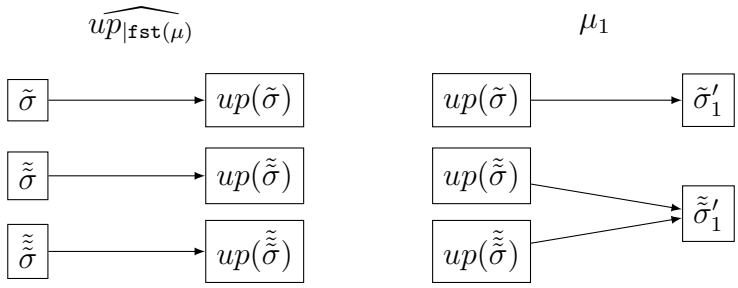
FIGURE 3.1 – Translation of Pretty-Big-Step to multisemantics

In order to derive an axiom, the multistate should be consistent with the ax function : for every pair (σ, σ') of the multistate, $ax(\sigma) = \sigma'$. We forbid μ to be

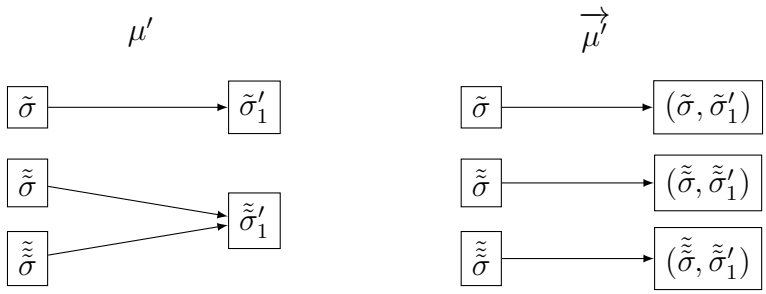
empty because it would correspond to multiderivations that have no meaning. In fact, we will see in chapter 4 that it could even induce wrong annotations.

Deriving a rule 1 with some μ can be done if μ_1 is such that for every pair (σ, σ') in the multistate μ, there exists a state σ_1 such that $up(\sigma) = \sigma_1$ and (σ_1, σ') is a pair of the multistate μ_1. The composition is strict so every state of $\mathtt{fst}(\mu_1)$ must be of the form $up(\sigma)$ with $\sigma \in \mathtt{fst}(\mu)$.

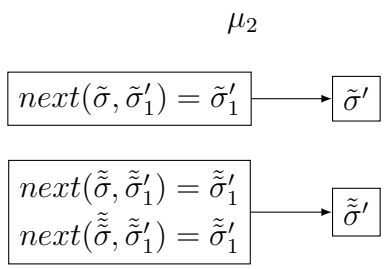
To derive a rule 2, we illustrate with a μ such that $fst(\mu) = \{\tilde{\sigma}, \tilde{\tilde{\sigma}}, \tilde{\tilde{\tilde{\sigma}}}\}$. We introduce a multistate μ' relating the semantic context of the derivation with the results of the first premise. The strict composition enforces that every state of $\mathtt{fst}(\mu_1)$ must be of the form $up(\sigma)$ with $\sigma \in \mathtt{fst}(\mu)$ and also that for every $\sigma \in \mathtt{fst}(\mu)$, $up(\sigma) \in \mathtt{fst}(\mu_1)$. We choose to illustrate μ_1 as if t_1 had the same behaviour in the states $up(\tilde{\tilde{\sigma}})$ and $up(\tilde{\tilde{\tilde{\sigma}}})$



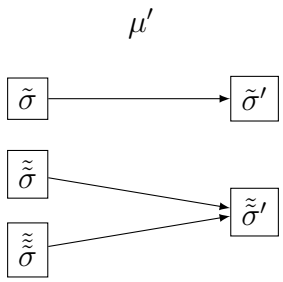
Thus μ' and $\overrightarrow{\mu'}$ are



Then, every state of $\mathtt{fst}(\mu_2)$ must be of the form $next(\sigma, \sigma'_1)$ with $(\sigma, \sigma'_1) \in \mathtt{snd}(\overrightarrow{\mu'})$ and also for every $(\sigma, \sigma'_1) \in \mathtt{snd}(\overrightarrow{\mu'})$, $next(\sigma, \sigma'_1) \in \mathtt{fst}(\mu_2)$. We choose to illustrate μ_2 in a case where $next(\sigma, \sigma'_1) = \sigma'_1$.



Finally, μ is the multistate



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.

$$\textsc{MltIf}\ \frac{\begin{array}{c} b \Downarrow \mu_1 \\ \texttt{If}_1\ s_1\ s_2 \Downarrow \mu_2 \qquad \mu' = \widehat{up_{|\texttt{fst}(\mu)}} \circ \mu_1 \qquad \mu = \overrightarrow{\mu'} \circ \widehat{next_{|\texttt{snd}(\overrightarrow{\mu'})}} \circ \mu_2 \end{array}}{\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \mu}$$

To derive $\texttt{If}_1\ s_1\ s_2$ there are two options. Either

$$\textsc{MltIfTrue}\ \frac{s_1 \Downarrow \mu_1 \qquad \mu = \widehat{up_{|\texttt{fst}(\mu)}} \circ \mu_1}{\texttt{If}_1\ s_1\ s_2 \Downarrow \mu}$$

where $\widehat{up_{|\texttt{fst}(\mu)}} = \{((M, true), M) \mid (M, true) \in \texttt{fst}(\mu)\}$ and then $\texttt{fst}(\mu)$ only contains states of the form $(M, true)$, or

$$\textsc{MltIfFalse}\ \frac{s_2 \Downarrow \mu_2 \qquad \mu = \widehat{up_{|\texttt{fst}(\mu)}} \circ \mu_2}{\texttt{If}_1\ s_1\ s_2 \Downarrow \mu}$$

where $\widehat{up_{|\texttt{fst}(\mu)}} = \{((M, false), M) \mid (M, false) \in \texttt{fst}(\mu)\}$ and then $\texttt{fst}(\mu)$

only contains states of the form $(M, false)$.

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, one may thus use two subderivations, one for each status of the guard, and merge them together.

$$\textsc{Merge}\ \frac{t \Downarrow \mu_1 \qquad t \Downarrow \mu_2}{t \Downarrow \mu_1 \cup \mu_2}$$

FIGURE 3.2 – The Merge rule

We do not restrict the use of the MERGE rule but in practice, we only use it when we need to apply different rules to a multistate. For example the evaluation of a conditional could look like the following derivation.

$$\textsc{MltIf}\ \frac{b \Downarrow \mu_1 \qquad \textsc{Merge}\ \dfrac{\dfrac{s_1 \Downarrow \mu'_{true}}{\texttt{If}_1\ s_1\ s_2 \Downarrow \mu_{true}} \qquad \dfrac{s_2 \Downarrow \mu'_{false}}{\texttt{If}_1\ s_1\ s_2 \Downarrow \mu_{false}}}{\texttt{If}_1\ s_1\ s_2 \Downarrow \mu_2}}{\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \mu}$$

with the conditions :

$$\begin{aligned}
\mu &= \overrightarrow{\mu'} \circ \widehat{next}_{|\texttt{snd}(\overrightarrow{\mu'})} \circ \mu_2 \\
\mu' &= \widehat{up}_{|\texttt{fst}(\mu)} \circ \mu_1 \\
\mu_2 &= \mu_{true} \cup \mu_{false} \\
\mu_{true} &= \widehat{up}_{|\texttt{fst}(\mu_{true})} \circ \mu'_{true} \\
\mu_{false} &= \widehat{up}_{|\texttt{fst}(\mu_{false})} \circ \mu'_{false}
\end{aligned}$$

where each up and $next$ function is the one corresponding to the rule where the condition appears.

3.3 Properties

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

Lemma 1. $\forall t \mu.\ t \Downarrow \mu \implies \forall (\sigma, \sigma') \in \mu.\ \sigma, t \to \sigma'$

The converse implication is not true, however. Consider the following program

```
n := In "argument";
i := 0;
While (i<n) do
  i := i + 1
```

This an example of a program allowing PBS derivations of arbitrary size. For every $k \in \mathbb{N}$, a derivation starting with the value k in the input `"argument"` needs to run k times the while loop. Each of these derivations is finite, but considering all of them for $k \in \mathbb{N}$ together would require an infinite multiderivation.

Nonetheless, when taking a finite number of PBS derivations, we are able to derive them all together in the multisemantics. 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 it using Lemmas 2 and 3. The first one states that if a term is derivable in PBS then it is derivable in the multisemantics with the corresponding singleton relation. The second lemma states that if a term is derivable with two multistates then it is derivable with the union of them.

Lemma 2. $\forall t, \sigma, \sigma'.\ \sigma, t \to \sigma' \implies t \Downarrow \{(\sigma, \sigma')\}$

Lemma 3. $\forall t, \mu_1, \mu_2.\ t \Downarrow \mu_1 \implies t \Downarrow \mu_2 \implies t \Downarrow \mu_1 \cup \mu_2$

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).

3.4 Multisemantics of WHILE

To illustrate a multisemantics, figures 3.3, 3.4 and 3.5 show the multisemantic rules for the WHILE language defined in section 1.3.3. These rules are automatically derived from the PBS and do not require to be defined by hand.

To simplify the reading, the conditions on the multistates are simplified as follow. In most cases, up is the identity function on its domain. Therefore $\widehat{up_{|\mathtt{fst}(\mu)}}$ is the identity relation on $\mathtt{fst}(\mu)$ and thus when $\widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1$ is defined

$$\widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1 = \mu_1$$

If this is the case for a rule 1, then $\mu = \mu_1$. And if this is the case for a rule 2, then $\mu' = \mu_1$.

Also, in some cases $next$ is a function always returning the second argument (the result of the first premise) and ignoring the first one (the semantic context of the conclusion) therefore we can simplify

$$\overrightarrow{\mu'} \circ \widehat{next_{|\mathtt{snd}(\mu')}} = \mu'$$

If this is the case for a rule 2, then $\mu = \mu' \circ \mu_2$.

Figure 3.3 gathers the multisemantics rules for the expressions.

The MLTCST rule to derive a constant c requires μ to be a multistate relating memories with states containing the same memory and only c in the extra. Deriving a variable x, the rule MLTVAR is similar to the previous rule : it requires μ to relate memories with the same memory paired with the value of x in this memory.

In the rule MLTPLUS, up is the identity function and $next$ returns the second argument, we can then simplify the equalities with the above remarks and the condition on μ becomes $\mu = \mu_1 \circ \mu_2$. It makes sense since the semantic context of the first premise is exactly the same as the conclusion and the semantic context of the second premise is the result of the first premise. The case of the MLTPLUS1 is more complex because the rule ignores the extra

$$\textsc{MltCst}\ \frac{\mu = \{(M, (M, c)) \mid M \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{c \Downarrow \mu}$$

$$\textsc{MltVar}\ \frac{\mu = \{(M, (M, M(x))) \mid M \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{x \Downarrow \mu}$$

$$\textsc{MltPlus}\ \frac{e_1 \Downarrow \mu_1 \qquad +_1\ e_2 \Downarrow \mu_2 \qquad \mu = \mu_1 \circ \mu_2}{e_1 + e_2 \Downarrow \mu}$$

$$\textsc{MltPlus1}\ \frac{\begin{array}{c} e_2 \Downarrow \mu_1 \qquad +_2 \Downarrow \mu_2 \qquad \mu' = \{((M, v_1), M) \mid (M, v_1) \in \mu\} \circ \mu_1 \\ \mu = \overrightarrow{\mu'} \circ \{(((M, v_1), (M', v_2)), (M', [v_1, v_2])) \mid ((M, v_1), (M', v_2)) \in \texttt{snd}(\overrightarrow{\mu'})\} \circ \mu_2 \end{array}}{+_1\ e_2 \Downarrow \mu}$$

$$\textsc{MltPlus2}\ \frac{\mu = \{((M, [v_1, v_2]), (M, v_1 + v_2)) \mid (M, [v_1, v_2]) \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{+_2 \Downarrow \mu}$$

$$\textsc{MltInput}\ \frac{\mu = \{(M, (M, M(i))) \mid M \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{\texttt{Input}\ i \Downarrow \mu}$$

FIGURE 3.3 – Rules of the WHILE multisemantics (expressions)

in the first premise and reconsiders it in the second premise. The semantic context of the first premise is not the one of the conclusion anymore ; moreover the semantic context of the second premise is not the result of the first premise either. In that case μ' is the composition of a relation forgetting the extra (defined on the same first part as μ) and μ_1. Then μ is the composition of three relations :

- $\overrightarrow{\mu'}$, the same relation as μ' but with the first part remembered in the second part ;
- $\{(((M, v_1), (M', v_2)), (M', [v_1, v_2])) \mid ((M, v_1), (M', v_2)) \in \mathtt{snd}(\overrightarrow{\mu'})\}$, a relation relating a pair of states with the same state as the second one but with the extra of the first state added to the extra of the second one ;
- and μ_2.

To finalize the derivation of an addition, the rule MLTPLUS2 simply requires μ to relate states containing two elements in the extra with states containing the sum of those elements in the extra.

The last rule to derive an expression is the MLTINPUT rule. As for variable, μ requires to relate memories with the same memory paired to the value of i in this memory.

Figure 3.4 gathers the multisemantics rules for a first subset of the statements.

The term `skip` is derived thanks to the rule MLTSKIP requiring μ to be a non-empty identity relation defined at most over states with empty extra.

For the sequence, MLTSEQ and MLTSEQ1 verify the remarks we made about up being identity functions and $next$ returning the second argument, therefore the equalities are straightforward.

The MLTIF is also straighforward because of the same remark. The two rules to handle both possible continuations are similar : MLTIFTRUE requires μ to be the strict composition of

- a relation relating states containing $true$ in the extra with the same state without extra,
- and the multistate μ_1.

$$\textsc{MltSkip}\ \frac{\mu = \{(M, M) \mid M \in \mathtt{fst}(\mu)\} \qquad \mu \neq \emptyset}{\mathtt{skip} \Downarrow \mu}$$

$$\textsc{MltSeq}\ \frac{s_1 \Downarrow \mu_1 \qquad ;_1\, s_2 \Downarrow \mu_2 \qquad \mu = \mu_1 \circ \mu_2}{s_1; s_2 \Downarrow \mu} \qquad \textsc{MltSeq1}\ \frac{s \Downarrow \mu}{;_1\, s \Downarrow \mu}$$

$$\textsc{MltIf}\ \frac{e \Downarrow \mu_1 \qquad \mathtt{If}_1\ s_1\ s_2 \Downarrow \mu_2 \qquad \mu = \mu_1 \circ \mu_2}{\mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \Downarrow \mu}$$

$$\textsc{MltIfTrue}\ \frac{s_1 \Downarrow \mu_1 \qquad \mu = \{((M, true), M) \mid (M, true) \in \mathtt{fst}(\mu)\} \circ \mu_1}{\mathtt{If}_1\ s_1\ s_2 \Downarrow \mu}$$

$$\textsc{MltIfFalse}\ \frac{s_2 \Downarrow \mu_1 \qquad \mu = \{((M, false), M) \mid (M, false) \in \mathtt{fst}(\mu)\} \circ \mu_1}{\mathtt{If}_1\ s_1\ s_2 \Downarrow \mu}$$

$$\textsc{MltWhile}\ \frac{e \Downarrow \mu_1 \qquad \mathtt{while}_1\ e\ \mathtt{do}\ s \Downarrow \mu_2 \qquad \mu = \mu_1 \circ \mu_2}{\mathtt{while}\ e\ \mathtt{do}\ s \Downarrow \mu}$$

$$\textsc{MltWhileFalse}\ \frac{\mu = \{((M, false), M) \mid (M, false) \in \mathtt{fst}(\mu)\} \qquad \mu \neq \emptyset}{\mathtt{while}_1\ e\ \mathtt{do}\ s \Downarrow \mu}$$

$$\textsc{MltWhileTrue1}\ \frac{\begin{array}{c} s \Downarrow \mu_1 \qquad \mathtt{while}_2\ e\ \mathtt{do}\ s \Downarrow \mu_2 \\ \mu = \{((M, true), M) \mid (M, true) \in \mathtt{fst}(\mu)\} \circ \mu_1 \circ \mu_2 \end{array}}{\mathtt{while}_1\ e\ \mathtt{do}\ s \Downarrow \mu}$$

$$\textsc{MltWhileTrue2}\ \frac{\mathtt{while}\ e\ \mathtt{do}\ s \Downarrow \mu}{\mathtt{while}_2\ e\ \mathtt{do}\ s \Downarrow \mu}$$

FIGURE 3.4 – Rules of the WHILE multisemantics (statements)

while MLTIFFALSE has the same requirement when $true$ is replaced by $false$.

The rule MLTWHILE is straighforward since up and $next$ are how we described them at the beginning of this section. To stop the while loop, the axiom MLTWHILEFALSE forces μ to be a multistate relating state containing $false$ in the extra with the same state but containing an empty extra. In the rule MLTWHILETRUE1, the $next$ function can be simplified as in the remark above. μ must then be the composition of

- a relation relating states containing $false$ in the extra with the same state without extra,
- μ_1,
- and μ_2.

The MLTWHILETRUE2 rule also has an identity up function, therefore the rule is totally simplified.

Figure 3.5 gathers the remaining multisemantics rules for the statments.

The MLTASG rule also has an identity up function since the expression is derived in the same semantic context as the assignment. Moreover the semantic context of the continuation is the result of the derivation of the expression, thus we have $\mu = \mu_1 \circ \mu_2$ again. The continuation of this rule is the axiom MLTASG1 which requires the multistate to be a relation between a state containing a single value v and the same state with the variable x set to v and an empty extra.

The MLTOUTPUT and MLTOUTPUT1 rules are analogous to the previous MLTASG and MLTASG1 rules, except that the store does not erase previous values.

Finally, the rule MERGE allows to gather two multiderivation of the same term that may have different structures.

3.5 Conclusion

In this chapter we gave a framework to build a new semantics when given an initial semantics in Pretty-Big-Step form. This new semantic derives

many executions of the same terms at once, and they share the same derivation rule when they are at the same program point. This last property gives us good hope in annotating the multisemantics in order to capture non-interference.

$$\textsc{MltAsg}\ \frac{e \Downarrow \mu_1 \qquad x :=_1 \Downarrow \mu_2 \qquad \mu = \mu_1 \circ \mu_2}{x := e \Downarrow \mu}$$

$$\textsc{MltAsg1}\ \frac{\mu = \{((M, v), M[x \mapsto v]) \mid (M, v) \in \mathtt{fst}(\mu)\} \qquad \mu \neq \emptyset}{x :=_1 \Downarrow \mu}$$

$$\textsc{MltOutput}\ \frac{e \Downarrow \mu_1 \qquad \mathtt{Ouput}_1\ o \Downarrow \mu_2 \qquad \mu = \mu_1 \circ \mu_2}{\mathtt{Ouput}\ o\ e \Downarrow \mu}$$

$$\textsc{MltOutput1}\ \frac{\mu = \{((M, v), M[o \mapsto v :: M(o)]) \mid (M, v) \in \mathtt{fst}(\mu)\} \qquad \mu \neq \emptyset}{\mathtt{Ouput}_1\ o \Downarrow \mu}$$

$$\textsc{Merge}\ \frac{t \Downarrow \mu_1 \qquad t \Downarrow \mu_2 \qquad \mu = \mu_1 \cup \mu_2}{t \Downarrow \mu}$$

FIGURE 3.5 – Rules of the WHILE multisemantics (statements) (bis)

CHAPITRE 4

ANNOTATIONS

The multisemantics is a tool able to reason about many derivation with one particularity : it can collect pieces of information from each derivation and aggregate them at the same program point. This chapter focuses on annotating this semantics to capture the non-interference property and states the correctness theorem of these annotations. This approach is not limited to non-interference, it applies to any hyperproperty that requires a finite number of derivations. We conjecture this methodology could be used to capture properties like nonmalleable non-interference [15] which needs four derivations.

4.1 Construction

The annotations track the inputs on which every variable and output depends in a dependency environment of type $MemDep$, typically written D. Additionally, we track the *context dependency* CD of the current computation. It has type $CtxtDep$, a set of inputs, and it represents the dependency on the context in which the current expression or statement is evaluated. The context dependency is used to track indirect flows like program counter levels do.

$$MemDep := (Var \cup Outputs) \mapsto Inputs\ set \qquad CtxtDep := Inputs\ set$$

An annotated derivation of a term t in a multistate μ is written

$$CD, D, t \Downarrow \mu, D', VD'$$

where $CD \in CtxtDep$ is the context dependency and $D \in MemDep$ is the environment dependency before the execution. $D' \in MemDep$ is the envi-

ronment dependency after the execution of the term. $VD' \in CtxtDep$ is the set of inputs the computed value, i.e., the extra, depends on. We will refer to CD and D as the entering dependencies and to D' and VD' as resulting dependencies. The way to interpret such a statement is "given entering dependencies CD and D, the derivation of t in the multistate μ returns the resulting dependencies D' and VD'".

As stated in section 2.2.1, we suppose we are given, for each axiom, 4 sets representing the inputs, variables and outputs it may read or write and a parameter specifying if the axiom produces an extra. Formally, each axiom comes with four sets and a boolean value : $InputRead \subset Inputs$, the set of inputs the axiom may read ; $VarRead \subset Var$, the set of variables the axiom may read ; $VarWrite \subset Var$, the set of variables the axiom may write ; $OutputWrite \subset Outputs$, the set of outputs the axiom may write ; $ProduceExtra$, a boolean value specifying if the axiom produces an extra. Informally, the extra can be see as a variable : adding the extra in the set of the elements the axiom can write on corresponds to a binary choice represented by $ProdExtra$. This analogy also works for reading the extra, considering that an axiom always reads the extra there is no need to introduce another boolean.

In practice these sets are empty sets or singletons because axioms are generally atomic operations modifying and using a small part of the memory. When having an axiom $Ax\ t\ ax\ InputRead\ VarRead\ VarWrite\ OutputWrite\ ProdExtra$, these parameters respect the following hypotheses.

1. If two states agree on the extra, $InputRead$, and $VarRead$, then for every $x \in VarWrite$, the value in x after the axiom is the same in both states.

   Hypothesis 1. *Membership of VarWrite*
   $\forall \sigma_1 \sigma_2,$

$$\begin{cases} extra(\sigma_1) = extra(\sigma_2) \\ \sigma_1(InputRead) = \sigma_2(InputRead) \\ \sigma_1(VarRead) = \sigma_2(VarRead) \end{cases} \implies \forall x \in VarWrite, ax(\sigma_1)(x) = ax(\sigma_2)(x)$$

2. Every variable x such that $x \notin VarWrite$ is not modified by the axiom.

   Hypothesis 2. *Non-membership of VarWrite*
   $\forall \sigma,$
   $$\forall x \notin VarWrite, \sigma(x) = ax(\sigma)(x)$$

3. If two states agree on the extra, $InputRead$, and $VarRead$, then for every $o \in OutWrite$, the values added to o after the axiom are the same in both states.

   Hypothesis 3. *Membership of OutWrite*
   $\forall \sigma_1 \sigma_2,$
   $$\begin{cases} extra(\sigma_1) = extra(\sigma_2) \\ \sigma_1(InputRead) = \sigma_2(InputRead) \\ \sigma_1(VarRead) = \sigma_2(VarRead) \end{cases} \implies \forall o \in OutWrite, \exists lv, \begin{cases} ax(\sigma_1)(o) = lv@\sigma_1(o) \\ ax(\sigma_2)(o) = lv@\sigma_2(o) \end{cases}$$

4. Every output o such that $o \notin OutWrite$ is not modified by the axiom.

   Hypothesis 4. *Non-membership of OutWrite*
   $\forall \sigma,$
   $$\forall x \notin OutWrite, \sigma(o) = ax(\sigma)(o)$$

5. If two states agree on the extra, $InputRead$, and $VarRead$, then if an extra is produced, the resulting extra is the same in both states.

   Hypothesis 5. *Producing an extra*
   $\forall \sigma_1 \sigma_2,$
   $$\begin{cases} extra(\sigma_1) = extra(\sigma_2) \\ \sigma_1(InputRead) = \sigma_2(InputRead) \\ \sigma_1(VarRead) = \sigma_2(VarRead) \\ prodExtra \end{cases} \implies extra(ax(\sigma_1)) = extra(ax(\sigma_2))$$

6. If no extra is produced, then it should be an empty list.

   Hypothesis 6. *Not producing an extra*
   $\forall \sigma,$
   $$\neg prodExtra \implies extra(ax(\sigma)) = nil$$

These hypotheses are non-interference-like properties over the axioms represented in terms of dependencies. Combining the annotations correctly in the derivation will allow to lift that property over all the programs.

Hypotheses 2, 4 and 6 could be more precise and enforce the implications to be double implications. We could rewrite them

$$\forall x \in Var, x \notin VarWrite \iff \forall \sigma, \sigma(x) = ax(\sigma)(x)$$

$$\forall o \in Outputs, o \notin OutputWrite \iff \forall \sigma, \sigma(o) = ax(\sigma)(o)$$

$$\neg prodExtra \iff \forall \sigma, extra(ax(\sigma)) = nil$$

Not verifying the right-to-left implications is not critical in terms of correctness ; but it may induce a loss of precision in the annotations.

The annotated semantics rules in Figure 4.1 are the multisemantics rules extended with annotation information.

Axioms The most complex case is the one for axioms. First consider the set Dep_{local} of inputs involved in the axiom : it is the union of the current context dependency, the inputs the axiom may read, and the dependencies of the variables the axiom may read. For every variable written by the axiom, we replace the dependency for that variable by Dep_{local}. Note that this is a strong update : we throw away prior dependencies for that variable as it is overwritten. In contrast, for every output written by the axiom, we keep the old dependencies of the output and simply add Dep_{local}. The dependency of the computed value is then Dep_{local} provided that there is one (which is specified by $produceExtra$). If no extra is produced the value dependency set is empty.

Rule 1 Rules 1 are simple to annotate : they propagate annotations. The entering dependencies of the premise are exactly the entering dependencies of the conclusion and the resulting dependencies of the conclusion are exactly the resulting dependencies of the premise.

$$\textsf{AMLTAX}\ \frac{\mu = \widehat{ax_{|\mathtt{fst}(\mu)}} \qquad \mu \neq \emptyset}{CD, D, t \Downarrow \mu, D', VD'}$$

where $Dep_{local} = CD \cup InputRead \bigcup_{x \in VarRead} D(x)$

$$VD' = \begin{cases} Dep_{local} & \text{if } ProduceExtra \\ \{\} & \text{otherwise} \end{cases},$$

$$\forall x.\ D'(x) = \begin{cases} Dep_{local} & \text{if } x \in VarWrite \\ D(x) & \text{otherwise} \end{cases},$$

and $\forall o.\ D'(o) = \begin{cases} Dep_{local} \cup D(o) & \text{if } o \in OutputWrite \\ D(o) & \text{otherwise} \end{cases}$.

$$\textsf{AMLTR1}\ \frac{CD, D, t_1 \Downarrow \mu_1, D_1, VD_1 \qquad \mu = \widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1}{CD, D, t \Downarrow \mu, D_1, VD_1}$$

$$\textsf{AMLTR2}\ \frac{\begin{array}{cc} CD, D, t_1 \Downarrow \mu_1, D_1, VD_1 & CD \cup VD_1, D_1, t_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu' = \widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1 & \mu = \overrightarrow{\mu'} \circ \widehat{next_{|\mathtt{snd}(\overrightarrow{\mu'})}} \circ \mu_2 \end{array}}{CD, D, t \Downarrow \mu, D_2, VD_2}$$

$$\textsf{AMERGE}\ \frac{CD, D, t \Downarrow \mu_1, D_1, VD_1 \qquad CD, D, t \Downarrow \mu_2, D_2, VD_2}{CD, D, t \Downarrow \mu_1 \cup \mu_2, D', VD_1 \cup VD_2}$$

where $D'(xo) = D_1(xo) \cup D_2(xo)$ for all variable and output xo

FIGURE 4.1 – Types of rule for an annotated multisemantics

Rule 2 The annotations for a Rule 2 exhibits the need of the parameter $produceExtra$ of the axioms. First the entering dependencies are propagated to the first derivation. Then, the context dependency of the continuation is the union of the initial context dependency CD and the value dependency of the first derivation VD_1. Note that, independently from the fact that the first derivation produces an extra or not, all side effects happening during the derivation of t_1 are stored in D_1 and are taken into account in the continuation.

Let us take a closer look at the rule 2 to see where $produceExtra$ impacts the derivation.

$$\text{AMLTR2}\ \frac{\begin{array}{cc} CD, D, t_1 \Downarrow \mu_1, D_1, VD_1 & CD \cup VD_1, D_1, t_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu' = \widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1 & \mu = \overrightarrow{\mu'} \circ \widehat{next_{|\mathtt{snd}(\overrightarrow{\mu'})}} \circ \mu_2 \end{array}}{CD, D, t \Downarrow \mu, D_2, VD_2}$$

If the derivation of t_1 does not produce an extra and no over-approximation has been done when parameterizing $produceExtra$ in every axiom, then VD_1 is empty. This can be proved by induction on the annotated derivation of t_1 :

- In the case of an axiom, no approximation has been done, then $produceExtra$ is false. Therefore, $VD_1 = \emptyset$.
- In the case of a rule 1, the result is immediate by induction since t_1 produces an extra if and only if its premise produces an extra.
- In the case of a rule 2, the result is also immediate by induction on the second derivation since t_1 produces an extra if and only if its second premise produces an extra.
- In the case of a merge rule, the induction hypothesis ensures that the value dependencies are empty in both annotated derivation, therefore their union is empty too.

We can then conclude that VD_1 is empty which means that the context in which t_2 is derived only depends on CD as it is the case for the sequence rule for example.

$$\text{AMLTSEQ}\ \frac{\begin{array}{cc} CD, D, s_1 \Downarrow \mu_1, D_1, \{\} & \mathbf{CD}, D_1, ;_1 s_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu' = \widehat{up_{|\mathtt{fst}(\mu)}} \circ \mu_1 & \mu = \overrightarrow{\mu'} \circ \widehat{next_{|\mathtt{snd}(\overrightarrow{\mu'})}} \circ \mu_2 \end{array}}{CD, D, s_1; s_2 \Downarrow \mu, D_2, VD_2}$$

On the other hand, if the derivation produces an extra, then the dependencies of that extra VD_1 are added to the context dependencies to evaluate the continuation. An example of the second case, where an extra is produced, is the rule for conditionals.

$$\textsc{AMltIf}\ \frac{\begin{array}{c} CD, D, b \Downarrow \mu_1, D_1, VD_1 \qquad \mathbf{CD} \cup \mathbf{VD_1}, D_1, \texttt{If}_1\ s_1\ s_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu' = \widehat{up}_{|\texttt{fst}(\mu)} \circ \mu_1 \qquad \mu = \overrightarrow{\mu'} \circ \widehat{next}_{|\texttt{snd}(\overrightarrow{\mu'})} \circ \mu_2 \end{array}}{CD, D, \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \mu, D_2, VD_2}$$

Merge rule Finally the Merge rule simply merges the dependencies together by doing the point-wise union of the environment dependencies, 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.

4.2 Annotated multisemantics of WHILE

The annotated rules of the WHILE multisemantics are the same as for the simple multisemantics with additional information on the conclusion of the rules. The premises stay the same. All the rules 1 and rules 2 are annotated exactly like in the formal definition of the annotated multisemantics to stay as modular as possible. For example, the rule AMLTSEQ could be simplified because our WHILE language cannot produce an extra in the first term s_1. This can be seen by looking at the $next$ function for this rule and noticing that $next$ is only defined if the resulting state of the first derivation has an empty extra. We could then take the remark we made previously and simplify the rule. But adding the exception mechanism to the language would give the possibility to produce a non empty extra in the derivation of s_1 as we saw in section 1.3.2 with the rule ERR.

Figure 4.2 gathers the annotated multisemantics rules for the expressions.

When deriving a constant, the memory is left untouched and the extra (the value of the constant in that case) depends directly on the context. This

$$\textsf{AMLTCST}\ \frac{\mu = \{(M,(M,c)) \mid M \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{CD, D, c \Downarrow \mu, D, CD}$$

$$\textsf{AMLTVAR}\ \frac{\mu = \{(M,(M,M(x))) \mid M \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{CD, D, x \Downarrow \mu, D, CD \cup D(x)}$$

$$\textsf{AMLTPLUS}\ \frac{\begin{array}{c} CD, D, e_1 \Downarrow \mu_1, D_1, VD_1 \\ CD \cup VD_1, D_1, +_1\ e_2 \Downarrow \mu_2, D_2, VD_2 \qquad \mu = \mu_1 \circ \mu_2 \end{array}}{CD, D, e_1 + e_2 \Downarrow \mu, D_2, VD_2}$$

$$\textsf{AMLTPLUS1}\ \frac{\begin{array}{c} CD, D, e_2 \Downarrow \mu_1, D_1, VD_1 \qquad CD \cup VD_1, D_1, +_2 \Downarrow \mu_2, D_2, VD_2 \\ \mu' = \{((M,v_1),M) \mid (M,v_1) \in \mu\} \circ \mu_1 \\ \mu = \overrightarrow{\mu'} \circ \{(((M,v_1),(M',v_2)),(M',[v_1,v_2])) \mid ((M,v_1),(M',v_2)) \in \texttt{snd}(\overrightarrow{\mu'})\} \circ \mu_2 \end{array}}{CD, D, +_1\ e_2 \Downarrow \mu, D_2, VD_2}$$

$$\textsf{AMLTPLUS2}\ \frac{\begin{array}{c} \mu = \{((M,[v_1,v_2]),(M,v_1+v_2)) \mid (M,[v_1,v_2]) \in \texttt{fst}(\mu)\} \\ \mu \neq \emptyset \end{array}}{CD, D, +_2 \Downarrow \mu, D, CD}$$

$$\textsf{AMLTINPUT}\ \frac{\mu = \{(M,(M,M(i))) \mid M \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{CD, D, \texttt{Input}\ i \Downarrow \mu, D, CD \cup \{i\}}$$

FIGURE 4.2 – Rules of the WHILE annotated multisemantics (expressions)

is why the rule AMLTCST, when given a context dependency CD and an environment dependency D, returns the same environment dependency and the context dependency as value dependency. When deriving the rule AMLTVAR with a variable x, the memory is also untouched and the extra depends on the context and on every input x depends on.

To derive an addition, the rules AMLTPLUS and AMLTPLUS1 are not axioms so their annotations are generic. The rule AMLTPLUS2, like AMLTCST, does not depend on any variable or input and does not modify the environment therefore the environment dependency is the same and the value dependency is the context dependency.

The rule MLTINPUT only produces an extra and this extra only depends on the input read. As for variable, environment dependencies do not change and the value dependency is the context dependency plus the input read.

Figure 4.3 gathers the annotated multisemantics rules for a first subset of the statements.

The term `skip` does exactly nothing : the memory is not modified and no extra is produced. Thus, the resulting environment dependency is the same as the entering environment dependency and the value dependency is empty.

The sequence, the if statement and the while statement are handled by rules 2 and rules 1 with their generic annotations excepted for the rule AMLTWHILEFALSE. This rule does nothing more than checking that the extra is $false$ but this has no impact on the dependencies. The environment dependencies stay the same and the value dependency is empty.

Figure 4.4 gathers the remaining annotated multisemantics rules for the statements.

The rule AMLTASG is a rule 2 and has generic annotations. Its continuation AMLTASG1 is an axiom that modifies the content of the variable x and does not return any extra. The value of x now depends only on the current context. The resulting environment dependency is the previous one for which the dependency associated to x has been changed to the current context

$$\textsc{AMltSkip}\ \frac{\mu = \{(M, M) \mid M \in \mathtt{fst}(\mu)\} \qquad \mu \neq \emptyset}{CD, D, \mathtt{skip} \Downarrow \mu, D, \emptyset}$$

$$\textsc{AMltSeq}\ \frac{\begin{array}{c} CD, D, s_1 \Downarrow \mu_1, D_1, VD_1 \\ CD \cup VD_1, D_1, ;_1\, s_2 \Downarrow \mu_2, D_2, VD_2 \end{array} \qquad \mu = \mu_1 \circ \mu_2}{CD, D, s_1; s_2 \Downarrow \mu, D_2, VD_2}$$

$$\textsc{AMltSeq1}\ \frac{CD, D, s \Downarrow \mu, D_1, VD_1}{CD, D, ;_1\, s \Downarrow \mu, D_1, VD_1}$$

$$\textsc{AMltIf}\ \frac{\begin{array}{c} CD, D, e \Downarrow \mu_1, D_1, VD_1 \\ CD \cup VD_1, D_1, \mathtt{If}_1\ s_1\ s_2 \Downarrow \mu_2, D_2, VD_2 \end{array} \qquad \mu = \mu_1 \circ \mu_2}{CD, D, \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \Downarrow \mu, D_2, VD_2}$$

$$\textsc{AMltIfTrue}\ \frac{\begin{array}{c} CD, D, s_1 \Downarrow \mu_1, D_1, VD_1 \\ \mu = \{((M, true), M) \mid (M, true) \in \mathtt{fst}(\mu)\} \circ \mu_1 \end{array}}{CD, D, \mathtt{If}_1\ s_1\ s_2 \Downarrow \mu, D_1, VD_1}$$

$$\textsc{AMltIfFalse}\ \frac{\begin{array}{c} CD, D, s_2 \Downarrow \mu_1, D_1, VD_1 \\ \mu = \{((M, false), M) \mid (M, false) \in \mathtt{fst}(\mu)\} \circ \mu_1 \end{array}}{CD, D, \mathtt{If}_1\ s_1\ s_2 \Downarrow \mu, D_1, VD_1}$$

$$\textsc{AMltWhile}\ \frac{\begin{array}{c} CD, D, e \Downarrow \mu_1, D_1, VD_1 \\ CD \cup VD_1, D_1, \mathtt{while}_1\ e\ \mathtt{do}\ s \Downarrow \mu_2, D_2, VD_2 \end{array} \qquad \mu = \mu_1 \circ \mu_2}{CD, D, \mathtt{while}\ e\ \mathtt{do}\ s \Downarrow \mu, D_2, VD_2}$$

$$\textsc{AMltWhileFalse}\ \frac{\mu = \{((M, false), M) \mid (M, false) \in \mathtt{fst}(\mu)\} \qquad \mu \neq \emptyset}{CD, D, \mathtt{while}_1\ e\ \mathtt{do}\ s \Downarrow \mu, D, \emptyset}$$

$$\textsc{AMltWhileTrue1}\ \frac{\begin{array}{c} CD, D, s \Downarrow \mu_1, D_1, VD_1 \\ CD \cup VD_1, D_1, \mathtt{while}_2\ e\ \mathtt{do}\ s \Downarrow \mu_2, D_2, VD_2 \\ \mu = \{((M, true), M) \mid (M, true) \in \mathtt{fst}(\mu)\} \circ \mu_1 \circ \mu_2 \end{array}}{CD, D, \mathtt{while}_1\ e\ \mathtt{do}\ s \Downarrow \mu, D_2, VD_2}$$

$$\textsc{AMltWhileTrue2}\ \frac{CD, D, \mathtt{while}\ e\ \mathtt{do}\ s \Downarrow \mu, D_1, VD_1}{CD, D, \mathtt{while}_2\ e\ \mathtt{do}\ s \Downarrow \mu, D_1, VD_1}$$

FIGURE 4.3 – Rules of the WHILE annotated multisemantics (statements)

$$\textsf{AMLTASG}\ \frac{\begin{array}{c} CD, D, e \Downarrow \mu_1, D_1, VD_1 \\ CD \cup VD_1, D_1, x :=_1 \Downarrow \mu_2, D_2, VD_2 \qquad \mu = \mu_1 \circ \mu_2 \end{array}}{CD, D, x := e \Downarrow \mu, D_2, VD_2}$$

$$\textsf{AMLTASG1}\ \frac{\mu = \{((M, v), M[x \mapsto v]) \mid (M, v) \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{CD, D, x :=_1 \Downarrow \mu, D[x \mapsto CD], \emptyset}$$

$$\textsf{AMLTOUTPUT}\ \frac{\begin{array}{c} CD, D, e \Downarrow \mu_1, D_1, VD_1 \\ CD \cup VD_1, D_1, \texttt{Ouput}_1\ o \Downarrow \mu_2, D_2, VD_2 \qquad \mu = \mu_1 \circ \mu_2 \end{array}}{CD, D, \texttt{Ouput}\ o\ e \Downarrow \mu, D_2, VD_2}$$

$$\textsf{AMLTOUTPUT1}\ \frac{\mu = \{((M, v), M[o \mapsto v :: M(o)]) \mid (M, v) \in \texttt{fst}(\mu)\} \qquad \mu \neq \emptyset}{CD, D, \texttt{Ouput}_1\ o \Downarrow \mu, D[o \mapsto D(o) \cup CD], \emptyset}$$

$$\textsf{AMERGE}\ \frac{CD, D, t \Downarrow \mu_1, D_1, VD_1 \qquad CD, D, t \Downarrow \mu_2, D_2, VD_2}{CD, D, t \Downarrow \mu_1 \cup \mu_2, D', VD_1 \cup VD_2}$$

where $D'(xo) = D_1(xo) \cup D_2(xo)$ for all variable and output xo

FIGURE 4.4 – Rules of the WHILE annotated multisemantics (statements) (bis)

dependency and the value dependency is empty.

The rule AMLTOUTPUT1 is analogous to the previous rule AMLTASG1, except that, as the store does not erase previous values, the old dependency is not erased.

Finally, the rule AMERGE is exactly the same as the formal rule : the environment dependency is merged by doing the union variable by variable and output by output ; and the resulting value dependency is the union of both resulting value dependencies.

4.3 Capturing masking

We have re-written the running example of Section 1.1.3 in the WHILE language and we will call it P.

```
x := true;
y := true;
if Input "i"
    then x := false
    else skip;
if x
    then y := false
    else skip;
Output "o" y
```

We now show how our approach captures the dependency in that case. The non-interference property only needs two executions to be negated, we thus consider a multistate μ with two semantic contexts, one with $false$ in the first input and one with $true$. We derive the running example in the annotated multisemantics with empty entering dependencies and we show that the output o depends on the input i. We write $D_\emptyset$ the empty environment dependency, a function returning an empty set for every variable and output.

We detail here, step by step, the crucial points of the derivation.

The first `if` statement will be derived in states containing $true$ in the x and y variables. We first have to derive the condition `Input` 1 and then derive the

continuation. For the purpose of explanation, we replace the resulting dependencies with question marks and we will see how the rules will constrain the annotations to capture the information flow.

$$\frac{\emptyset, D_\emptyset, \texttt{Input}\ i \Downarrow \mu_i, D_\emptyset, \{i\} \qquad \dfrac{\{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu'_{P_1}, ?, ?}{\{i\}, D_\emptyset, ;_1\ \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu'_{P_1}, ?, ?}}{\emptyset, D_\emptyset, P_1 \Downarrow \mu_{P_1}, ?, ?}$$

with $\mu_{P_1} = \mu_i \circ \mu'_{P_1}$

The first premise obviously produces an extra (the value in the input i) and then its value dependency is $\{i\}$. The rule of the if statement makes the dependency flow into the context dependency of the continuation.

Unfortunately, both semantic contexts of μ'_{P_1} do not have the same extra because it corresponds to the value in the input i. The rule AMLTIFTRUE cannot be applied here and we need to use the merge rule to separate μ'_{P_1} into two multistates μ_t and μ_f.

$$\frac{\{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu_t, ?, ? \qquad \{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu_f, ?, ?}{\{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu'_{P_1}, D_x, \emptyset}$$

with $\mu'_{P_1} = \mu_t \cup \mu_f$.

The first premise goes into the first branch of the conditional and captures the dependency in $D_x = D_\emptyset[x \mapsto \{i\}]$ because the current context depends on the input i and an assignment has been done on x. The second branch does not have the dependency since a `skip` statement has been derived.

$$\frac{\dfrac{\{i\}, D_\emptyset, x := f \Downarrow \mu_t, D_x, \emptyset}{\{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu_t, D_x, \emptyset} \qquad \dfrac{\{i\}, D_\emptyset, \texttt{skip} \Downarrow \mu_f, D_\emptyset, \emptyset}{\{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu_f, D_\emptyset, \emptyset}}{\{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu'_{P_1}, D_x, \emptyset}$$

But the merge rule merges the dependencies and thus we know this piece of code may induce information leaking from input i to variable x. We should also notice that in the multistate μ'_{P_1} (and thus μ_{P_1}) the result having $true$

stored in the input i has the value $false$ stored in x and the other one has $true$ stored in x.

$$\dfrac{\emptyset, D_\emptyset, \texttt{Input}\ i \Downarrow \mu_i, D_\emptyset, \{i\} \qquad \dfrac{\{i\}, D_\emptyset, \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu'_{p_1}, D_x, \emptyset}{\{i\}, D_\emptyset, ;_1\ \texttt{If}_1\ x := f\ \texttt{skip} \Downarrow \mu'_{p_1}, D_x, \emptyset}}{\emptyset, D_\emptyset, P_1 \Downarrow \mu_{P_1}, D_x, \emptyset}$$

It leads us to derive P_2 with the following entering annotations.

$$\dfrac{\emptyset, D_x, x \Downarrow \mu_x, D_x, \{i\} \qquad \dfrac{\{i\}, D_x, \texttt{If}_1\ y := f\ \texttt{skip} \Downarrow \mu'_{P_2}, ?, ?}{\{i\}, D_x, ;_1\ \texttt{If}_1\ y := f\ \texttt{skip} \Downarrow \mu'_{P_2}, ?, ?}}{\emptyset, D_x, P_2 \Downarrow \mu_{P_2}, ?, ?}$$

with $\mu_{P_2} = \mu_x \circ \mu'_{P_2}$

After the derivation of the variable x, we know that the returned value depends on i because we derived x with the entering environment dependency D_x. Then the same phenomenon than previously appears, we need the merge rule to derive the left-hand part because the derivation of x made the extras of the semantic contexts of μ'_{P_2} to be different.

$$\dfrac{\dfrac{\{i\}, D_x, y := f \Downarrow \mu'_t, D_{xy}, \emptyset}{\{i\}, D_x, \texttt{If}_1\ y := f\ \texttt{skip} \Downarrow \mu'_t, D_{xy}, \emptyset} \qquad \dfrac{\{i\}, D_x, \texttt{skip} \Downarrow \mu'_f, D_x, \emptyset}{\{i\}, D_x, \texttt{If}_1\ y := f\ \texttt{skip} \Downarrow \mu'_f, D_x, \emptyset}}{\{i\}, D_x, \texttt{If}_1\ y := f\ \texttt{skip} \Downarrow \mu'_{P_2}, D_{xy}, \emptyset}$$

with $\mu'_{P_2} = \mu'_t \cup \mu'_f$

The first derivation tree captures the flow of the context to y and updates the resulting environment dependency $D_{xy} = D_x[y \mapsto \{i\}] = D_\emptyset[x \mapsto \{i\}][y \mapsto \{i\}]$.

Finally, when deriving the output with these annotations, we have the dependency between o and i because we know that y depended on i.

$$\dfrac{\emptyset, D_{xy}, y \Downarrow \mu_y, D_{xy}, \{i\} \qquad \{i\}, D_{xy}, \texttt{Ouput}_1\ o \Downarrow \mu'_o, D_{xy}, \emptyset}{\emptyset, D_{xy}, \texttt{Ouput}\ o\ y \Downarrow \mu_o, D_{xy}, \emptyset}$$

with $\mu_o = \mu_y \circ \mu'_o$.

4.4 Limits

Over-approximations in the parameters

The first source of over-approximation comes from the parameters of the axioms. The WHILE language is simple enough to precisely define these parameters, but in the general case an axiom may write and read many inputs, variables or outputs.

For instance we could introduce a term $\texttt{Swap}\ x\ y$ taking two variables x and y as parameters and swapping their value in the memory. This term is derived in PBS with the axiom

$$\textsc{Swap}\ \frac{}{M, \texttt{Swap}\ x\ y \to M[x \mapsto M(y)][y \mapsto M(x)]}$$

this axiom reads and writes the variables x and y thus :

$$\begin{aligned} VarRead = VarWrite &= \{x, y\} \\ InputRead = OutputWrite &= \emptyset \\ ProdExtra &= false \end{aligned}$$

Let us suppose we have the following program.

```
x := In i;
y := In j;
swap x y
```

When deriving the swap instruction in the annotated multisemantic, we already know that x depends on input i and y depends on input j.

We could then obtain the following derivation

$$\frac{}{\emptyset, D, \texttt{Swap}\ x\ y \Downarrow \mu, D', \emptyset}$$

such that

$$D = D_\emptyset[x \mapsto i][y \mapsto j]$$
$$\text{and } D' = D_\emptyset[x \mapsto \{i, j\}][y \mapsto \{i, j\}].$$

The result is that, after the swap, both x and y depend on both i and j which is an overapproximation since now x does not depend on i anymore.

To fix this overapproximation, it seems sufficient to give, instead of the two sets $VarRead$ and $InputRead$, a set of inputs and a set of variables for each variable and each output in $VarWrite$ and $OutputWrite$.

Value-related overapproximations

We also may capture dependencies that do not lead to interference even if the axioms are perfectly parametrized. For example, any annotated multiderivation of the following program will conclude that the output o depends on the input i.

```
if Input i
  then Output o 1
  else Output o 1
```

The derivation has the same structure than the derivation of P_1 that we build in section 4.3. But in fact, changing input i would not change the output o.

The loss of precision comes from the fact that we only track dependencies, and not the actual values being computed.

Non-executable semantics

This semantic is clearly not executable because of the ability to derive an infinite number of single derivations. But this is not an issue for two reasons. The first one is that we are not trying to build an analyser but a mathematical object close to non-interference that is easier to manipulate in formal proof. The second reason is that anyway, we want to be as complete as possible and in the ideal case where there are no approximations, we could not be executable since TINI is undecidable.

CHAPITRE 5

CORRECTNESS

Now that we are convinced that the annotations capture non-interference, we intend to formally prove it. Establishing a link between the annotated multisemantics and the formal definition of non-interference will later allow to prove the correctness of analyzers. The correctness of the annotations is expressed as follows

Theorem 1 (Fundamental Theorem)**.** $\forall t$,
If $\forall \mu, D', VD', o$ public,
$D_\emptyset, \emptyset, t \Downarrow \mu, D', VD' \implies \forall i \in (D'o), i$ is public
Then t is non-interferent.

It states that if for all annotated multiderivation, any public output does not depend on a private input then the program is non-interferent. Of course, this theorem is valid under some hypotheses over the PBS rules. We will briefly explain the proof method and then we explain how the theorem is used in practice to prove analyzers.

5.1 Hypotheses

Determinism (2 hypotheses)

There are two notions of determinism we assume. The first one is a determinism over the choice of the rules when deriving a term and the second one is the determinism of the language.

The first hypotheses we make is that, given a term t and a state σ, the knowledge of the extra is enough to determine the unique rule that may be applied to t in the state σ. In another word, if two rules apply on two states agreeing on the extra, then the two rules are identical. The predicate `applies`

`r sigma` states, if the rule is an axiom (respectively rule 1 or rule 2), that `ax sigma <> None` (respectively `up sigma <> None`).

```
Hypothesis deterministic_rule :
  forall t sigma1 sigma2 r1 r2,
    List.In r1 (Rules t)
    -> List.In r2 (Rules t)
    -> applies r1 sigma1
    -> applies r2 sigma2
    -> extra sigma1 = extra sigma2
    -> r2 = r1
.
```

Moreover, the rules themselves are deterministic : deriving a term t in some state will always result in the same state.

```
Hypothesis deterministic_deriv :
  forall t sigma sigma1' sigma2',
    deriv t sigma sigma1'
    -> deriv t sigma sigma2'
    -> sigma1' = sigma2'
.
```

Axioms (6 hypotheses)

Concerning axioms, we assume the writer of the semantics gave correct parameters for $inputRead$, $varRead$, $varWrite$, $outputWrite$ and $prodExtra$. The following hypothesis are the formalization of hypotheses 1 to 6 of section 4.1

On one hand, if two states are equal on the extra and on the inputs and variables read by the axiom ($inputRead$ and $variableRead$), then applying an axiom on those two states results in states that are equal on the variable written by the axiom ($varWrite$). On the other hand, applying an axiom leaves unmodified the variables not in $varWrite$.

```
Hypothesis HypVarWrite :
  forall t ax iRead vRead vWrite oWrite prodExtra,
```

```
    List.In (Ax t ax iRead vRead vWrite oWrite prodExtra)
            (Rules t)
    -> forall sigma1 sigma2 sigma1' sigma2',
         (ax sigma1 = Some sigma1')
         -> (ax sigma2 = Some sigma2')
         -> (extra sigma1 = extra sigma2)
         -> (forall i, mem i iRead -> (Envi sigma1 i
                                        = Envi sigma2 i))
         -> (forall y, mem y vRead -> (Envx sigma1 y
                                        = Envx sigma2 y))
         -> (forall x, mem x vWrite -> (Envx sigma1' x
                                         = Envx sigma2' x))
.

Hypothesis HypNotVarWrite :
  forall t ax iRead vRead vWrite oWrite prodExtra,
    List.In (Ax t ax iRead vRead vWrite oWrite prodExtra)
            (Rules t)
    -> forall sigma sigma',
         ax sigma = Some sigma'
         -> forall x, x \notin vWrite
                      -> Envx sigma x = Envx sigma' x
.
```

An analogous reasoning can be done for the outputs : if two states are equal on the extra and on the inputs and variables read by the axiom ($inputRead$ and $varRead$), then applying an axiom on those two states results in states that have the same value in the first place of every output written by the axiom ($outputWrite$). Moreover, applying an axiom leaves unmodified the outputs not in $outputWrite$.

```
Hypothesis HypOutputWrite :
  forall t ax iRead vRead vWrite oWrite prodExtra,
    List.In (Ax t ax iRead vRead vWrite oWrite prodExtra)
            (Rules t)
```

```
    -> forall sigma1 sigma2 sigma1' sigma2',
         (ax sigma1 = Some sigma1')
         -> (ax sigma2 = Some sigma2')
         -> (extra sigma1 = extra sigma2)
         -> (forall i, mem i iRead -> (Envi sigma1 i
                                          = Envi sigma2 i))
         -> (forall y, mem y vRead -> (Envx sigma1 y
                                          = Envx sigma2 y))
         -> forall o,
              mem o oWrite
              -> exists lv,
                   (Envo sigma1' o = lv ++ (Envo sigma1 o))
                   /\ (Envo sigma2' o = lv ++ (Envo sigma2 o))
.

Hypothesis HypNotOutputWrite :
  forall t ax iRead vRead vWrite oWrite prodExtra,
    List.In (Ax t ax iRead vRead vWrite oWrite prodExtra)
            (Rules t)
    -> forall sigma sigma',
         (ax sigma = Some sigma')
         -> (forall o, notin o oWrite
                       -> (Envo sigma o = Envo sigma' o))
.
```

Finally, the read sets also have an impact on the extra : applying an axiom that produces an extra on two states that are equal on the initial extra and on the inputs and variables read by the axiom ($inputRead$ and $varRead$) results in states that have the same extra. Conversely, if the axiom does not produce an extra, then the resulting extra is empty.

```
Hypothesis HypExtraProduced :
  forall t ax iRead vRead vWrite oWrite prodExtra,
    List.In (Ax t ax iRead vRead vWrite oWrite prodExtra)
            (Rules t)
```

```
    -> forall sigma1 sigma2 sigma1' sigma2',
         (ax sigma1 = Some sigma1')
         -> (ax sigma2 = Some sigma2')
         -> (extra sigma1 = extra sigma2)
         -> (forall i, mem i iRead -> (Envi sigma1 i
                                          = Envi sigma2 i))
         -> (forall y, mem y vRead -> (Envx sigma1 y
                                          = Envx sigma2 y))
         -> (prodExtra = true)
         -> (extra sigma1' = extra sigma2')
.


Hypothesis HypNoExtraProduced :
  forall t ax iRead vRead vWrite oWrite prodExtra,
    List.In (Ax t ax iRead vRead vWrite oWrite prodExtra)
            (Rules t)
    -> forall sigma sigma',
      (ax sigma = Some sigma')
      -> ( (prodExtra = false)
           ->
           (extra sigma' = nil))
.
```

up and next (3 hypotheses)

We also give some constraints on the up and $next$ functions. We suppose that they do not impact the memory and that they may only change the extra. 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 : $up(M, e) = (M', e') \implies M' = M \wedge e' = f(e)$. For rules 2, if $next$ is defined, then the new memory is the memory of the second argument, and the new extra only depends on the extras of the arguments : $next((M_1, e_1), (M_2, e_2)) = (M, e) \implies M = M_2 \wedge e = g(e_1, e_2)$. Finally, given a term and an extra, at most one rule applies.

```
Hypothesis HypUpR1 :
  forall t t1 up,
    List.In (R1 t t1 up) (Rules t)
    -> exists f_extra,
       forall sigma sigma',
         (up sigma = Some sigma')
         -> sameMemory sigma sigma'
            /\ extra sigma' = f_extra (extra sigma)
.

Hypothesis HypUpR2 :
  forall t t1 t2 up next,
    List.In (R2 t t1 t2 up next) (Rules t)
    -> exists f_extra,
       forall sigma sigma',
         (up sigma = Some sigma')
         -> sameMemory sigma sigma'
            /\ extra sigma' = f_extra (extra sigma)
.

Hypothesis HypNext :
  forall t t1 t2 up next,
    List.In (R2 t t1 t2 up next) (Rules t)
    -> exists g_extra,
       forall sigma sigma' sigma'',
         (next sigma sigma' = Some sigma'')
         -> sameMemory sigma' sigma''
            /\ extra sigma'' = g_extra (extra sigma)
                                       (extra sigma')
.
```

Inputs (1 hypothesis)

We also request that a derivation may not change the inputs of the state.

```
Hypothesis HypNoWriteInput :
  forall t s s',
    deriv t s s'
    -> forall i, (inputs s i = inputs s' i)
.
```

These hypotheses are not constraining except maybe for the determinism of the semantics. It forbids random generators but we can still model programs with a bounded number k of calls to a random generator by allocating k particular cells to this purpose before the execution of a program.

5.2 Correctness theorem

To compare the annotations and the non-interference property, we need to formally define non-interference for a given language in Pretty-Big-Step style as follow ( ).

We suppose we are given a language

```
Context {Syntax : AbstractSyntax}.
Context {Semantics : AbstractSemantics}.
```

and two functions to determine if inputs and outputs are public or not. An input or output that is not public will be considered to be private.

```
Parameter isPublicInput : input -> bool.
Parameter isPublicOutput : output -> bool.
```

We can now define non-interference as in section 1.1.1

```
Definition NonInterferent (t : Term) : Prop :=
  forall sigma1 sigma1' sigma2 sigma2',
    deriv t sigma1 sigma1'
    -> deriv t sigma2 sigma2'
    -> initialState sigma1
    -> initialState sigma2
    -> (forall i, Bool.Is_true (isPublicInput i)
         -> Envi sigma1 i = Envi sigma2 i)
```

```
    -> (forall o, Bool.Is_true (isPublicOutput o)
         -> Envo sigma1' o = Envo sigma2' o)
.
```

We must specify that $\sigma 1$ and $\sigma 2$ are initial states, i.e. the variables are set to the same initial value and that the outputs are empty.

The main theorem we prove is the following. It states that a term t is non-interferent provided that for all annotated multiderivation of t starting with empty dependencies and for all public output o the derivation ends with public inputs in the dependency of o.

Theorem 2 (Fundamental Theorem 🐓)**.** $\forall t$,

If $\forall \mu, D', VD'$,

$D_\emptyset, \emptyset, t \Downarrow \mu, D', VD' \implies \forall o$ public, $\forall i \in (D'o), i$ is public

Then t is non-interferent.

Depending on the situation, it may be more interesting to use the contraposition :

Theorem 3 (Contraposition)**.** $\forall t$,

If t is interferent

Then $\exists \mu, D', VD'$ public such that

$D_\emptyset, \emptyset, t \Downarrow \mu, D', VD' \wedge \exists o$ public such that $\exists i \in (D'o), i$ is private

The corresponding coq code is the following (🐓)

```
Theorem correctness :
  forall t,
    (* if *)
    (forall mu D' VD' o,
    (* for all multiderivation  *)
       annot_multi_deriv empty (fun xo => empty) t mu D' VD'
         (* and for all public output o *)
       -> Bool.Is_true (isPublicOutput o)
         (* all the dependencies of o are public *)
       -> (forall i, mem i (D' (O o))
            -> Bool.Is_true(isPublicInput i))
```

```
    (* then t is non-interferent *)
  -> NonInterferent t
.
```

5.3 Proving an analyzer

Among the known methods to prove an analyzer, each one assumes the program variables are given a security level (public or private in the traditional case). One advantage of our technique is that we only need to give a security level to the inputs and outputs a program may take, and not the internal variables.

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_{NI}. It is a function returning $false$ for at least each interferent program and may have some false-negatives. But if the function returns $true$, it means the analyzed program doubtlessly satisfies the property of non-interference. In another words, if the program is interferent, A_{NI} must reject the program. The correctness property for the analyzer A_{NI} is the following :

Lemma 4. $\forall P$, if $A_{NI}(P)$ then P is non-interferent.

Such proofs may be difficult to do by induction on 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, it is sufficient to prove :

Lemma 5. $\forall P$,
If $A_{NI}(P)$

Then $\forall \mu, D', VD', i$ private

$$D_\emptyset, \emptyset, P \Downarrow \mu, D', VD' \implies \forall o \text{ public}, \forall i \in (D'o), i \text{ is public}$$

As an example, figure 5.1 we consider the analyzer ( ) over our WHILE language. This naive analyzer rejects every program that outputs a value on a public output.

To prove the correctness of this analyzer, instead of using the notion of non-interference, we prove the intermediate lemma ( ) of Figure 5.2 by induction on the multiderivation. This lemma states that if the analyzer authorizes the program and if a multiderivation starts with empty dependencies for the public outputs, then the multiderivation ends with empty dependencies for the public outputs.

This lemma allows to easily prove lemma 5 and then ensure the correctness of the annotations.

```
Fixpoint NoPublicOutputAnalyzer (t : Term) :=
  match t with
      | Skip => true
      | Seq s1 s2 => andb (NoPublicOutputAnalyzer s1)
                          (NoPublicOutputAnalyzer s2)
      | Seq1 s2 => NoPublicOutputAnalyzer s2
      | If e st sf => andb (NoPublicOutputAnalyzer e)
                           (andb (NoPublicOutputAnalyzer st)
                                 (NoPublicOutputAnalyzer sf))
      | If1 st sf => (andb (NoPublicOutputAnalyzer st)
                           (NoPublicOutputAnalyzer sf))
      | While e s => andb (NoPublicOutputAnalyzer e)
                          (NoPublicOutputAnalyzer s)
      | While1 e s => andb (NoPublicOutputAnalyzer e)
                           (NoPublicOutputAnalyzer s)
      | While2 e s => andb (NoPublicOutputAnalyzer e)
                           (NoPublicOutputAnalyzer s)

      | Assign x e => (NoPublicOutputAnalyzer e)
      | Assign1 x => true
      | Out o e => andb (negb (isPublicOutput o))
                        (NoPublicOutputAnalyzer e)
      | Out1 o => negb (isPublicOutput o)

      | Var x => true
      | Cons v => true
      | Plus e1 e2 => andb (NoPublicOutputAnalyzer e1)
                           (NoPublicOutputAnalyzer e2)
      | Plus1 e2 => (NoPublicOutputAnalyzer e2)
      | Plus2 => true
      | In i => true
    end.
```

FIGURE 5.1 – A non-interference analyzer

```
Lemma invariant :
  forall t CD D mu D' VD',
    annot_multi_deriv CD D t mu D' VD'
    -> Is_true (NoPublicOutputAnalyzer t)
    -> (forall o , Is_true (isPublicOutput o)
                   -> (D (O o) = \{}))
    -> (forall o , Is_true (isPublicOutput o)
                   -> (D' (O o) = \{}))
.
```

FIGURE 5.2 – Intermediate lemma

CONCLUSION

Summary

With the increasing use of software in our daily life and in particular with sensitive data, managing the information flow and formally ensuring security properties becomes necessary. In this thesis, we proposed a way to automatically build a new semantics (the annotated multisemantics) when an initial one is given in Pretty-Big-Step form. The annotated multisemantics is a formal object collecting local information of many classic executions and merging this information to detect interferent programs. This new semantics is a formally proved tool to prove non-interference analyzers.

Pretty-Big-Step

The construction of this new semantics is automatic, granted that the original one is in PBS style. Additionally to the inherent advantages of PBS, it ensures a structure easy to formalize. The axioms of the PBS semantics must come with some extra parameters ensuring a sort of non-interference over them only. Then this PBS must also verify 12 hypotheses :

1. Determinism over the rules : given a term and a state, only one rule applies ;
2. Determinism over the results : given a term, a state and a rule, only one state can be the result ;
3. 6 hypotheses concerning the correctness of additional parameters of the axioms ;
4. 3 hypotheses over the up and $next$ functions ;
5. and one final hypothesis ensuring that inputs never change.

Annotated Multisemantics

From this Pretty-Big-Step semantics, we are then able to automatically

build a multisemantics able to reason about many derivation at once and to share information between two or more classic derivation at some program point. Annotating this multiderivation allows us to track the dependencies during the considered executions and we proved that these annotations are correct in relation to the non-interference property : if, among all the multiderivations of a program, none of them can exhibit a public output depending on a private input then the program is non-interferent.

Proving an analyzer

Finally we formally proved the correctness of the annotations in relation to the notion of non-interference.

Theorem 4 (Fundamental Theorem 🐓)**.** $\forall t$,

If $\forall \mu, D', VD'$,

$D_\emptyset, \emptyset, t \Downarrow \mu, D', VD' \implies \forall o$ public, $\forall i \in (D'o), i$ is public

Then t is non-interferent.

The theorem allows to prove the correctness of analyzers by reasoning only with multiderivations which are defined by induction, unlike non-interference which is harder to manipulate.

Perspectives

This work can be continued in various directions. We already stated that it is possible to improve the precision of the annotations. It would be interesting to look at different hyperproperties and see if other annotated multisemantics could also capture them. Finally, there is room for improvement in the path of proving a analyzer for JavaScript.

Precision of the annotations

When writing the axioms rules in PBS, instead of the two sets $VarRead$ and $InputRead$, it may be interesting to give a set of input and a set of variable for each variable and each output in $VarWrite$ and $OutputWrite$. It should improve the precision for axioms like Swap (see section 4.4) but this will not remove value-dependent approximations.

Toward completeness

Even if the problem of non-interference is undecidable, our multisemantics can try to reach completeness since it only is a mathematical object for proofs and not an executable semantics. To achieve completeness it seems mandatory to investigate into the actual values stored or generated.

Different hyperproperty

Another direction to look into is the formalization of another hyperproperty, especially ones requiring more than two executions to define, as nonmalleable information flow [15] for example.

Toward JavaScript

Finally, proving a JavaScript analyzer would be a huge step and it requires at least to rewrite the axioms of the Pretty-Big-Step semantics of JavaScript already existing, to adapt the memory model and to prove the 12 hypotheses.

BIBLIOGRAPHIE

[1] http://heartbleed.com/, 2014.

[2] Torben AMTOFT, Sruthi BANDHAKAVI et Anindya BANERJEE, « A logic for information flow in object-oriented programs », in : *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, 2006, p. 91–102, DOI : 10.1145/1111037.1111046, URL : http://doi.acm.org/10.1145/1111037.1111046.

[3] Aslan ASKAROV et al., « Termination-Insensitive Noninterference Leaks More Than Just a Bit », in : *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, 2008, p. 333–348.

[4] Mounir ASSAF et al., « Hypercollecting semantics and its application to static analysis of information flow », in : *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, 2017, p. 874–887, URL : http://dl.acm.org/citation.cfm?id=3009889.

[5] Thomas H. AUSTIN et Cormac FLANAGAN, « Efficient Purely-dynamic Information Flow Analysis », in : *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, Dublin, Ireland : ACM, 2009, p. 113–124, ISBN : 978-1-60558-645-8, DOI : 10.1145/1554339.1554353, URL : http://doi.acm.org/10.1145/1554339.1554353.

[6] Thomas H. AUSTIN et Cormac FLANAGAN, « Multiple facets for dynamic information flow », in : *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, 2012, p. 165–178.

[7] Thomas H. AUSTIN et Cormac FLANAGAN, « Permissive Dynamic Information Flow Analysis », in : *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, Toronto, Canada : ACM, 2010, 3 :1–3 :12, ISBN : 978-1-60558-827-8, DOI : 10.1145/1814217.1814220, URL : http://doi.acm.org/10.1145/1814217.1814220.

[8] Michael BACKES, Boris KÖPF et Andrey RYBALCHENKO, « Automatic Discovery and Quantification of Information Leaks », in : *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, 2009, p. 141–153, DOI : 10.1109/SP.2009.18, URL : https://doi.org/10.1109/SP.2009.18.

[9] Gilles BARTHE, Pedro R. D'ARGENIO et Tamara REZK, « Secure information flow by self-composition », in : *Mathematical Structures in Computer Science* 21.*6* (2011), p. 1207–1252.

[10] Gilles BARTHE, David PICHARDIE et Tamara REZK, « A Certified Lightweight Non-interference Java Bytecode Verifier », in : *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, 2007, p. 125–140, DOI : 10.1007/978-3-540-71316-6_10, URL : https://doi.org/10.1007/978-3-540-71316-6_10.

[11] Lennart BERINGER et Martin HOFMANN, « Secure information flow and program logics », in : *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, 2007, p. 233–248, DOI : 10.1109/CSF.2007.30, URL : https://doi.org/10.1109/CSF.2007.30.

[12] Nataliia BIELOVA et Tamara REZK, « A Taxonomy of Information Flow Monitors », in : *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016, p. 46–67.

[13] Martin BODIN et al., « A Trusted Mechanised JavaScript Specification », in : *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*, San Diego, CA, USA, jan. 2014, p. 87–100.

[14] Gérard BOUDOL et Ilaria CASTELLANI, « Noninterference for Concurrent Programs », in : *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, 2001, p. 382–395, DOI : 10.1007/3-540-48224-5_32, URL : https://doi.org/10.1007/3-540-48224-5_32.

[15] Ethan CECCHETTI, Andrew C. MYERS et Owen ARDEN, « Nonmalleable Information Flow Control », in : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, p. 1875–1891, DOI : 10.1145/3133956.3134054, URL : http://doi.acm.org/10.1145/3133956.3134054.

[16] Arthur CHARGUÉRAUD, « Pretty-Big-Step Semantics », in : *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, 2013, p. 41–60.

[17] Michael R. CLARKSON et Fred B. SCHNEIDER, « Hyperproperties », in : *Journal of Computer Security* 18.*6* (2010), p. 1157–1210.

[18] Ellis COHEN, « Information Transmission in Computational Systems », in : *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, West Lafayette, Indiana, USA : ACM, 1977, p. 133–139.

[19] Thierry COQUAND et Gérard HUET, « The calculus of constructions », thèse de doct., INRIA, 1986.

[20] Dorothy E. DENNING, « A Lattice Model of Secure Information Flow », in : *Commun. ACM* 19.*5* (1976), p. 236–243, DOI : 10.1145/360051.360056, URL : http://doi.acm.org/10.1145/360051.360056.

[21] Dorothy E. DENNING et Peter J. DENNING, « Certification of Programs for Secure Information Flow », in : *Commun. ACM* 20.*7* (juil. 1977), p. 504–513, ISSN : 0001-0782.

[22] Dominique DEVRIESE et Frank PIESSENS, « Noninterference through Secure Multi-execution », in : *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, p. 109–124.

[23] Joseph A. GOGUEN et José MESEGUER, « Security Policies and Security Models », in : *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, 1982, p. 11–20.

[24] Georges GONTHIER, « The Four Colour Theorem : Engineering of a Formal Proof », in : *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, 2007, p. 333, DOI : 10.1007/978-3-540-87827-8_28, URL : https://doi.org/10.1007/978-3-540-87827-8%5C_28.

[25] Georges GONTHIER et al., « A Machine-Checked Proof of the Odd Order Theorem », in : *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, 2013, p. 163–179, DOI : 10.1007/978-3-642-39634-2_14, URL : https://doi.org/10.1007/978-3-642-39634-2%5C_14.

[26] Daniel HEDIN et Andrei SABELFELD, « Information-Flow Security for a Core of JavaScript », in : *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, 2012, p. 3–18, DOI : 10.1109/CSF.2012.19, URL : https://doi.org/10.1109/CSF.2012.19.

[27] Sebastian HUNT et David SANDS, « On flow-sensitive security types », in : *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, 2006, p. 79–90.

[28] Vineeth KASHYAP, Ben WIEDERMANN et Ben HARDEKOPF, « Timing- and Termination-Sensitive Secure Information Flow : Exploring a New Approach », in : *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, 2011, p. 413–428.

[29] Gurvan LE GUERNIC, « Confidentiality Enforcement Using Dynamic Information Flow Analyses », thèse de doct., Kansas State University, 2007, URL : http://tel.archives-ouvertes.fr/tel-00198621/fr/.

[30] Xavier LEROY, « Formal verification of a realistic compiler », in : *Communications of the ACM* 52.*7* (2009), p. 107–115, URL : http://xavierleroy.org/publi/compcert-CACM.pdf.

[31] Jacques-Louis LIONS et al., *Flight 501 Failure*, http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, 1996.

[32] Chris NEWCOMBE et al., « How Amazon web services uses formal methods », in : *Commun. ACM* 58.*4* (2015), p. 66–73, DOI : 10.1145/2699417, URL : http://doi.acm.org/10.1145/2699417.

[33] Andrei POPESCU, Johannes HÖLZL et Tobias NIPKOW, « Proving Concurrent Noninterference », in : *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, 2012, p. 109–125, DOI : 10.1007/978-3-642-35308-6_11, URL : https://doi.org/10.1007/978-3-642-35308-6_11.

[34] Alejandro RUSSO et Andrei SABELFELD, « Dynamic vs. Static Flow-Sensitive Security Analysis », in : *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, 2010, p. 186–199.

[35] Andrei SABELFELD et Andrew C. MYERS, « Language-based information-flow security », in : *IEEE Journal on Selected Areas in Communications* 21.*1* (2003), p. 5–19.

[36] Andrei SABELFELD et Alejandro RUSSO, « From Dynamic to Static and Back : Riding the Roller Coaster of Information-Flow Control Research », in : *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, 2009, p. 352–365.

[37] Matthieu SOZEAU et Nicolas OURY, « First-Class Type Classes », in : *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Procee-*

dings, 2008, p. 278–293, DOI : 10.1007/978-3-540-71067-7_23, URL : https://doi.org/10.1007/978-3-540-71067-7%5C_23.

[38] The Coq development TEAM, *The Coq proof assistant reference manual*, Version 8.6, 2016, URL : http://coq.inria.fr.

[39] Philip WADLER et Stephen BLOTT, « How to Make ad-hoc Polymorphism Less ad-hoc », in : *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, 1989, p. 60–76, DOI : 10.1145/75277.75283, URL : http://doi.acm.org/10.1145/75277.75283.

ANNEXE A

PBS RULES OF WHILE IN COQ

```
Inductive Concr_value : Type :=
| Num (z:Z)
| Bool (b:bool)
.

Inductive Concr_Term: Type :=
(*Statements*)
| Skip : Concr_Term
| Seq : Concr_Term -> Concr_Term -> Concr_Term
| Seq1 : Concr_Term -> Concr_Term
| If : Concr_Term -> Concr_Term -> Concr_Term -> Concr_Term
| If1 : Concr_Term -> Concr_Term -> Concr_Term
| While : Concr_Term -> Concr_Term -> Concr_Term
| While1 : Concr_Term -> Concr_Term -> Concr_Term
| While2 : Concr_Term -> Concr_Term -> Concr_Term
| Assign : variable -> Concr_Term -> Concr_Term
| Assign1 : variable -> Concr_Term
| Out : output -> Concr_Term -> Concr_Term
| Out1 : output -> Concr_Term

(*Expressions*)
| Var : variable -> Concr_Term
| Cons : Concr_value -> Concr_Term
| Plus : Concr_Term -> Concr_Term -> Concr_Term
| Plus1 : Concr_Term  -> Concr_Term
| Plus2 : Concr_Term
| In : input -> Concr_Term
```

.

```
Definition Concr_Rules (t:Concr_Term): list rule :=
  match t with
    (*Statements*)
    | Skip =>
        (Ax Skip
            (fun sc => match extra sc with
                       | nil => Some sc
                       | _ => None end)
            empty empty empty empty
            false)
      :: nil

    | Seq s1 s2 =>
        (R2 (Seq s1 s2)
            (s1)
            (Seq1 s2)
            (fun sc => match extra sc with
                       | nil => Some sc
                       | _ => None end)
            (fun sc res => match extra sc, extra res with
                           | nil, nil => Some res
                           | _, _ => None end))
      :: nil

    | Seq1 s2 =>
        (R1 (Seq1 s2)
            (s2)
            (fun sc => match extra sc with
                       | nil => Some sc
                       | _ => None end))
      :: nil
```

```
| While e s =>
    (R2 (While e s)
        (e)
        (While1 e s)
        (fun sc => match extra sc with
                   | nil => Some sc
                   | _ => None end)
        (fun sc res => match extra sc, extra res with
                       | nil, b :: nil => Some res
                       | _,_ => None end))
  :: nil

| While1 e s =>
    (R2 (While1 e s)
        (s)
        (While2 e s)
        (fun sc => (match extra sc with
                    | (Bool true) :: nil =>
                          Some (update_extra sc nil)
                    | _ => None end))
        (fun sc res => match extra sc, extra res with
                       | (Bool true) :: nil, nil =>
                            Some res
                       | _,_ => None end))
 :: (Ax (While1 e s)
        (fun sc => (match extra sc with
                    | (Bool false) :: nil =>
                          Some (update_extra sc nil)
                    | _ => None end))
        empty empty empty empty
        false)
 :: nil

| While2 e s =>
```

```
    (R1 (While2 e s)
        (While e s)
        (fun sc => match extra sc with
                   | nil => Some sc
                   | _ => None end))
 :: nil

| Assign x e =>
    (R2 (Assign x e)
        (e)
        (Assign1 x)
        (fun sc => match extra sc with
                   | nil => Some sc
                   | _ => None end)
        (fun sc res => match extra sc, extra res with
                       | nil, v :: nil => Some res
                       | _,_ => None end))
 :: nil

| Assign1 x =>
    (Ax (Assign1 x)
        (fun sc => match extra sc with
                   | v::nil =>
                      Some (update_extra
                              (update_var sc x v)
                              nil)
                   | _ => None end)
        empty empty (singleton x) empty
        false)
 :: nil

| Out1 o =>
    (Ax (Out1 o)
        (fun sc => match extra sc with
```

```
                    | v::nil =>
                          Some (update_extra
                                  (update_output sc o v)
                                  nil)
                    | _ => None end)
        empty empty empty (singleton o)
        false)
 :: nil

| Out o e=>
    (R2 (Out o e)
        (e)
        (Out1 o)
        (fun sc => match extra sc with
                    | nil => Some sc
                    | _ => None end)
       (fun sc res => match extra sc, extra res with
                       | nil, v :: nil => Some res
                       | _,_ => None end))
 :: nil

| If e s1 s2 =>
    (R2 (If e s1 s2)
        (e)
        (If1 s1 s2)
        (fun sc => match extra sc with
                    | nil => Some sc
                    | _ => None end)
        (fun sc res => match extra sc, extra res with
                       | nil, (Bool _) :: nil =>
                          Some res
                       | _,_ => None end))
 :: nil
```

```
| If1 s1 s2 =>
    (R1 (If1 s1 s2)
        (s1)
        (fun sc => (match extra sc with
                    | (Bool true) :: nil =>
                          Some (update_extra sc nil)
                    | _ => None end)))
 :: (R1 (If1 s1 s2)
        (s2)
        (fun sc => (match extra sc with
                    | (Bool false) :: nil =>
                          Some (update_extra sc nil)
                    | _ => None end)))
 :: nil

(*Expressions*)
| Cons v =>
    (Ax (Cons v)
        (fun sc => match extra sc with
                   | nil =>
                     Some (update_extra sc (v::nil))
                   | _ => None end)
        empty empty empty empty
        true)
 :: nil

| Var x =>
    (Ax (Var x)
        (fun sc => match extra sc, Envx sc x with
                   | nil, Some v =>
                     Some (update_extra sc (v :: nil))
                   | _,_ => None end)
        empty (singleton x) empty empty
        true)
```

```
 :: nil

| Plus e1 e2 =>
    (R2 (Plus e1 e2)
        (e1)
        (Plus1 e2)
        (fun sc => match extra sc with
                   | nil => Some sc
                   | _ => None end)
        (fun sc res => match extra sc, extra res with
                       | nil, (Num _)::nil => Some res
                       | _,_ => None end))
 :: nil

| Plus1 e2 =>
    (R2 (Plus1 e2)
        (e2)
        (Plus2)
        (fun sc => match extra sc with
                   | (Num _)::nil =>
                        Some (update_extra sc nil)
                   | _ => None end)
        (fun sc res => match extra sc, extra res with
                       | (Num n1)::nil, (Num n2)::nil =>
                            Some (update_extra res
                                  [(Num n1);(Num n2)])
                       | _,_ => None end))
 :: nil

| Plus2 =>
    (Ax (Plus2)
        (fun sc => match extra sc with
                   | (Num n1) :: (Num n2) :: nil =>
                        Some (update_extra sc
```

```
                                    [Num (n1+n2)])
                        | _ => None end)
            empty empty empty empty
            true)
    :: nil

  | In i =>
      (Ax (In i)
            (fun sc => match extra sc with
                        | nil =>
                          Some (update_extra sc [Envi sc i])
                        | _ => None end)
            (singleton i) empty empty empty
            true)
    :: nil
end.
```

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Certification d'Analyses Non Locales avec une Sémantique Annotée

Mot clés : multisémantique annotée ; non-interférence ; certification

Resumé : La quantité croissante de données traitées par les logiciels rend légitime le besoin de garanties de confidentialité. La propriété de non-interférence assure qu'un programme ne fuite pas de données privées vers une sortie publique.

Nous proposons une méthode pour construire, une multisémantique annotée capable de capturer la propriété de non-interférence pour aider à prouver formellement des analyseurs. Nous fournissons un théorème prouvé indiquant que les annotations capturent correctement la non-interférence.

Le théorème de correction permet de prouver un analyseur sans s'appuyer sur la définition de non-interférence mais sur les annotations.

Title : Non Local Analyses Certification With an Annotated Semantics

Keywords : annotated multisemantics ; non-interference ; certification

Abstract : Because of the increasing quantity of data processed by software, the need for privacy guarantees is legitimate. The property of non-interference ensures that a program does not leak private data to a public output.

We propose a framework to build an annotated multisemantics able to capture the non-interference property to help formally prove analysers. The framework comes with a proved theorem stating that the annotations correctly capture non-interference.

The correctness theorem allows to prove an analyser without relying on the definition of non-interference but on the annotations.