Université Paris 8
IUT de Montreuil
Département Informatique

Formalisation des garanties de sécurité apportées par l'isolation de composants logiciels

Inria de Paris - Équipe Prosecco

Rapport de stage du DUT Informatique, promotion 2015-2016

Stage du 11 Avril au 1er Juillet 2016

ENG SAMBO BORIS

Superviseurs : Rémi Georges (Enseignant référent)
Yannis Juglaret (Maître de stage)

Montreuil 2016

Résumé

Dans le cadre de ma formation en DUT Informatique, j'ai réalisé mon stage de fin d'études d'une durée de 12 semaines au sein du centre de recherche **Inria Paris** et plus précisément dans l'équipe-projet **Prosecco** *(Programming Securely with cryptography)*.

Je me suis orienté vers ce stage car je m'intéressais aux méthodes formelles et à la recherche dans le domaine de l'informatique. J'ai voulu découvrir ses mécanismes de l'intérieur afin de guider mes choix de carrière.

Inria est un institut de recherche national français travaillant sur les mathématiques et l'informatique. L'institut entretient de nombreuses relations avec les industries (Microsoft, Alstom...), est à l'origine de nombreuses start-ups (Cryptosense, Esterel Technologies...) et collabore avec de nombreuses universités et organismes de recherche (Université Paris-Sud, Université Paris Diderot, CNRS, CEA etc). De nombreux outils célèbres ont été conçus par ses chercheurs (Coq, OCaml, Scilab).

L'équipe *Prosecco* est quant à elle une équipe-projet assez jeune travaillant principalement sur l'utilisation des **méthodes formelles** pour la sécurité informatique.

Mon travail s'inscrit dans le cadre d'un **projet de recherche** de mon maître de stage et de d'autres chercheurs. Il s'agit de sécurité informatique formelle : simplement, un langage comme le langage C est **sensible** à certaines attaques liées à la mémoire par manque de vérifications statiques ou dynamiques (Buffer Overflow). Les programmes de ce langage sont compilés en un programme de langage de bas niveau. Le projet de recherche propose de montrer que dans un contexte particulier, les interactions possibles entre un attaquant et un programme de bas niveau ne vont pas au delà des interactions possibles dans le haut niveau. Et qu'ainsi, tout ce qui peut se passer sur un programme compilé est déjà prévu par la spécification (code source du programme). Cette propriété se nomme Compilation compartimentée sécurisée et représente le théorème principal.

Toutes les notions nécessaires et certaines preuves ont été formalisées mathématiquement sur papier, dans des fichiers textes.

Ma contribution correspond à une modélisation de ces concepts formels avec l'**assistant de preuves** *Coq*, la réalisation de certaines preuves et formaliser la nouvelle propriété avec Compilation Compartimentée Sécurisée avec Coq. Un assistant de preuves permet de programmer de façon classique mais aussi de programmer des preuves : on écrit la formulation d'un théorème et on construit sa preuve instructions par instructions. L'assistant se charge de confirmer que la preuve est correcte.

Avant de commencer, je me suis auto-formé sur l'assistant de preuve Coq. Pour cela je me suis servi du livre numérique *Software Foundations* dont 3 des auteurs travaillent dans le même projet que moi.

La première étape était la définition du **langage source** (qui avait déjà été conçu sur papier par mon maître de stage). Le langage source est un langage impératif très simple et proche du C utilisant des tableaux et des appels de procédures mais sans boucles explicites ni affectations de variables.

J'ai vérifié avec Coq que ce langage satisfaisait des propriétés classiques attendues.

La seconde étape consistait à définir un **langage de bas niveau** (langage assembleur) qui permet de manipuler directement notre propre modélisation de la mémoire et des registres (représentées par des listes d'entiers).

Pour relier nos deux langages j'ai défini un **compilateur** permettant de convertir les programmes écrits dans le langage source en programmes du langage cible.

Une fois que nous avons un langage d'exemple pour tester la propriété principale, il a fallu définir des outils nécessaires pour la prouver : la **sémantique des traces**, un formalisme mathématique permettant de modéliser le comportement d'un programme et d'un attaquant puis de raisonner dessus.

Et enfin, j'ai **formalisé la preuve du théorème principal** sur Coq ce qui a duré jusqu'à l'avant dernière semaine du stage. Le principe de la preuve avait déjà été présenté dans l'article, il fallait ensuite la "programmer" et la vérifier avec Coq pour s'assurer qu'elle soit correcte.

Pour la dernière semaine, il m'a été proposé par mon maître de stage et son directeur de thèse, d'assister à la conférence scientifique *CSF2016* à Lisbonne présentant les derniers travaux de la recherche en sécurité informatique.

Globalement, ce stage m'a permis de cerner l'environnement de travail de la recherche. J'ai pu assister à des séminaires, présentations, soutenances de thèse, lire des articles de recherche avec l'aide de mon maître de stage. J'ai aussi pu observer l'organisation des chercheurs et la rédaction d'un article.

Abstract

To complete my two-year technical degree in Computer Science, I have done my Internship within *Inria Paris* during 12 weeks and more precisely with the team **Prosecco** *(Programming Securely with cryptography)*.

This choice was mainly led by my interests in formal methods and research. I wanted to discover how research works in order to refine my career path.

Inria is a french national institute of research in Computer Science and Mathematics. It maintains a lot of relations with industries (Microsoft, Alstom...), is the source of many start-up companies (Cryptosense, Esterel Technologies...) and works jointly with a lot of other universities and research institutes (Université Paris-Sud, Université Paris Diderot, CNRS, CEA etc). Several famous pieces of software and languages are the result of the work of their researchers (Coq, OCaml, Scilab).

As for the team *Prosecco*, it's currently a quite young team. The team mainly works on the use of **formal methods** for computer security.

My work takes place in my supervisor's **research project**. The subject is formal computer security : basically, a C-like language can be really sensitive to memory-based vulnerabilities due to the lack of static or dynamic verifications. The programs are compiled from a source language to a target language (in a lower-level). One goal of the project is to prove (formally) that the possible interactions between an attacker and a compiled program are limited to those in the program written in the source language. Thus every possible actions on the compiled program should be already intended in the specification (source language). This property is called **Secure Compartmentalizing Compilation** and is the main theorem.

Every formal notations was already specified (in a mathematical form) on the paper and a text file.

During my internship, I had to tranpose the formal definitions into the *Coq* **proof assistant**, prove some lemmas and theorems and formalize the new Secure Compartmentalizing Compilation property into Coq. A proof assistant is a tool that allows to program as usual but also to program formal proofs. We have to write what we call the specification then the proof assistant check itself automatically if the proof is consistent.

To conclude, it was a rich experience : I could understand how the everyday life of researchers is (from Inria at least). I attended to a lot of seminars, thesis defenses, read academic papers (with the help of my supervisor). I could also see how researchers were organized and how they write papers.

Remerciements

Je tiens à remercier certaines personnes pour leur contribution directe ou indirecte à ce rapport de stage ou le soutien qu'ils m'ont apporté durant ce stage.

Je tiens à remercier **Yannis Juglaret**, mon maître de stage pour sa disponibilité, son écoute, ses efforts malgré mes très très nombreux mails envoyés et pour avoir nourri mon intérêt en me présentant de nouvelles choses qui sortaient parfois du cadre de mon stage.

Je tiens à remercier **Cătălin Hriţcu**, le directeur de thèse de Yannis pour le soutien qu'il m'a offert pour le domaine de la recherche (en intégrant mon nom dans l'article du projet dans lequel j'ai contribué) et pour ses efforts afin de m'intégrer à l'équipe.

Je tiens à remercier **Rémi Georges**, mon enseignant référent pour sa première visite à Inria qui a contribué à mon soutien.

Je tiens à remercier les relecteurs pour avoir pris le temps de lire ce rapport (même à un stade très expérimental) : **Adrien** et **Yoann**

Montreuil 2016

ENG Sambo Boris

Sommaire

Chapitre 1

Introduction

Afin de compléter ma formation en DUT Informatique à l'IUT de Montreuil, j'ai dû effectuer un stage d'une durée de 12 semaines. Ce stage constitue une première véritable expérience professionnelle afin de mettre en pratique ses compétences dans le domaine de l'informatique.

Étant intéressé par la recherche en informatique, après hésitations et contacts avec les enseignants-chercheurs de mon établissement j'ai choisi de poser ma candidature (parmi d'autres) dans l'institut de recherche *Inria de Paris* suite à une proposition de *Yannis Juglaret*, doctorant dans l'équipe *Prosecco*, au courant de mes intérêts qui devint ensuite mon maître de stage.

Il m'est proposé de contribuer à un projet de recherche en utilisant l'assistant de preuve Coq, un outil de méthodes formelles. Les méthodes formelles sont des méthodes utilisant la rigueur de la logique mathématiques sur des systèmes (informatiques, électroniques etc). Elles constituent un champs de recherche très active qui trouve de nombreuses applications industrielles (aéronautique, ferroviaire, aérospatial, des domaines où la sûreté est primordiale afin d'éviter des pertes matérielles et humaines) et puise ses sources dans des théorie très fructueuses (correspondance de Curry-Howard, théorie des types etc).

Dans ce document, je présente (I) l'organisme, le centre et l'équipe qui m'ont reçu, (II) ce qu'est un assistant de preuve et les objectifs du projet dans lequel j'ai travaillé, (III) le rôle que j'ai joué et les outils que j'ai utilisé, puis (IV) le travail que j'ai effectué orné de nombreuses explications et finalement (V) un bilan pour conclure sur mon stage avec notamment des commentaires subjectifs.

S'il vous plaît, référez-vous d'abord au glossaire pour découvrir les termes nécessaires à la compréhension du document et référez vous aux notations au fil de la lecture (à la fin du document).

Chapitre 2

Organisme d'accueil

L'*Institut national de recherche en informatique et en automatique* abrégé *Inria*, est un organisme français et public de recherche en mathématiques et en informatique. Il est rattaché au ministère de l'éducation nationale, de l'enseignement supérieur et de la recherche et au ministère chargé de l'industrie.

2.1 Inria, acteur de l'innovation technologique

2.1.1 Historique

L'*Inria* a été initialement fondé le 3 Mai 1967 sous le nom de *IRIA* pour *Institut de recherche en informatique et en automatique* dans le cadre du « *Plan Calcul* ».

Le « Plan Calcul » était un plan gouvernemental lancé en 1966 par le général *Charles de Gaulle*, *Michel Debré* et un groupe de hauts fonctionnaires afin de promouvoir les industries françaises et européennes en matière d'ordinateurs et d'inciter à l'émergence d'établissements d'enseignement et de recherche en informatique, une décision nécessaire face à l'opportunité qu'offrait l'ascension fulgurante des technologies de l'informatique.

FIGURE 2.1 – *Logo de l'INRIA jusqu'à 2011*

FIGURE 2.2 – *Logo de l'Inria à partir de 2011*

En 1979, l'*IRIA*, par le *décret du 27 Décembre*, devient l'*INRIA*, un organisme public à caractère administratif placé sous la tutelle du ministère de l'industrie.

Le 7 Juillet 2011, l'institut devient ce qu'il est aujourd'hui : *Inria*, le résultat d'un changement d'identité cosmétique marqué par la volonté d'abandonner l'acronyme associé à l'organisme (notamment à cause de son éloignement du domaine de l'automatique).

Actuellement, Inria connaît de fortes relations avec les industries et les universités. L'institut est à l'origine d'un grand nombre de start-up et de productions industrielles célèbres en informatique. Son président actuel est *Antoine Petit*.

2.1.2 Organisation

Au niveau stratégique, Inria adopte une stratégie qu'il intitule « Objectif 2020 » dans le but de contribuer aux défis de notre époque [1].

Au niveau géographique, Inria est constitué de **8 centres de recherches** répartis sur le territoire français : *Bordeaux, Grenoble, Lille, Nancy, Paris 12ème, Rennes, Saclay, Sophia Antipolis*.

Plusieurs **grands axes de recherche** sont communs à chaque centres mais chacun possède ses propres équipes. On retrouve les axes de recherche suivants :

- Mathématiques appliquées, calcul et simulation
- Algorithmique, programmation, logiciels et architectures
- Réseaux, systèmes et services, calcul distribué
- Perception, Cognition, Interaction
- Santé, biologie et planète numériques

2.1.3 Projets notables

Le travail de Inria couplé à celui de ses collaborateurs : les universités et le *CNRS (Centre National de Recherche Scientifique)* pour ne citer qu'eux, ont donné naissance à de nombreux projets de renommée :

- Le langage de programmation fonctionnelle **OCaml** utilisé par les universités françaises. C'est un des langages fonctionnels les plus utilisés actuellement.
- Le langage **Esterel** (créer par Gérard Berry), un langage synchrone et réactif utilisé pour les systèmes critiques (aéronautique, aérospatial, ferroviaire...).
- L'assistant de preuves **Coq** utilisé aussi bien pour les preuves formelles que pour les mathématiques. Il sera présenté dans ce document.
- L'outil de calcul numérique **Scilab**, une alternative libre et gratuite à Matlab utilisé pour l'enseignement des mathématiques.
- Le langage de programmation orienté objet **Pharo**
- L'assistant de preuve **F*** (F Star) encore dans un état assez embryonnaire avec la collaboration de *Microsoft Research*.

Inria, comme l'*Université Paris Diderot* et l'*Université Pierre et Marie Curie*, sont des fiers défenseurs du logiciel libre à l'origine de *IRILL (Initiative pour la recherche et l'innovation sur le logiciel libre)*, un laboratoire de recherche qui a pour objectif de favoriser le développement de logiciels libres.

2.1.4 Relations externes

Inria entretient des relations à la fois avec les établissements académiques et les industries. Elle encourage l'utilisation des connaissances académiques en industrie par les nombreuses start-up créées.

1. http ://www.inria.fr/institut/strategie/plan-strategique

Start-up créées : CryptoSense (outils de sécurité en cryptographie), Esterel Technologies (logiciels de conception pour systèmes fiables), Mnemotix (ingénierie des connaissances et web sémantique), ILog (leader mondial des composants logiciels C++ et Java racheté par IBM en 2009) et bien d'autres.

Partenaires industriels : Alcatel-Lucent, Alstom, Andra, EADS Astrium, EDF R&D, Google, Microsoft Research (relation assez forte qui a donné naissance au centre *Microsoft Research - Inria Joint Centre*[2]), Technicolor, Total.

Partenaires académiques : 30 accords signés avec des universités (Paris Sud, Joseph Fourier, Pierre et Marie Curie, Nice Sophia Antipolis, Paris Diderot, Bordeaux, Lille 1, Université des technologies de Troyes...), des grandes écoles et écoles d'ingénieur (Grenoble INP, ENS Lyon, ENS Cachan, Centrale Paris, Mines de Nantes...).

Instituts partenaires : Institut national de recherche en agronomie (INRA), Centre national de la recherche scientifique (CNRS), Centre d'étude du machinisme agricole et du génie rural des eaux et forêts (CEMAGREF), Centre français de recherche aérospatiale (ONERA), Fondation Agropolis, Commissariat à l'énergie atomique (CEA).

2.1.5 Chiffres clés (2015)

- Activité : **170** équipes-projets, **139** collaborations avec d'autres établissements
- Industries : **370** brevets, **143** logiciels déposés, 24 start-up créées
- Ressources humaines : **2700** collaborateurs (dont 1800 scientifiques) de **87** nationalités différentes
- Ressources économiques : **230 millions** d'euros de budget total

2.2 Inria de Paris, siège de l'institut

Le centre de Paris, autrefois appelé *Inria Paris - Rocquencourt* est le siège de l'institut. Sa directrice est Isabelle Ryl.

Le centre de Paris a déménagé début 2016 au 12ème arrondissement afin de se rapprocher de ses collaborateurs parisiens.

Plus précisément, le centre se situe dans le *Trio Daumesnil*, au 43 rue du Charolais (75012) à proximité de la Mairie du 12ème, de Gare de Lyon et de Bercy. Le *Trio Daumesnil* construit sur un ancien site de la SNCF, est au centre d'un projet ambitieux : celui de développer un nouveau quartier d'affaire qui s'étendra jusqu'à Gare de Lyon. Il est actuellement constitué de 3 bâtiments.

2.2.1 Équipes de recherche

Le centre de Paris possède de nombreuses équipes dont certaines sont délocalisées et situées dans des universités. Certaines autres sont des équipes « mixtes » réunissant des enseignant-chercheurs, des chargés de recherche du CNRS et de Inria.

2. http ://www.msr-inria.fr/

On peut citer les quelques équipes suivantes attachés aux centres de Paris pour la branche « Algorithmique, programmation, logiciels et architectures » mais il existe aussi des équipes dans les autres branches :

- $\pi.\mathbf{r}^2$ (Université Paris 7, CNRS, Inria). Conception, étude et implémentation de langages pour les preuves et les programmes. Rassemble les principaux développeurs de Coq. Sous-branche «Preuves et vérification».
- **Gallium** (Inria) Langages de programmation, types, compilation et preuves. Notamment responsables du langage OCaml. Sous-branche «Preuves et vérification».
- **PROSECCO** (Inria), Programming securely with cryptography. Sous-branche «Sécurité et confidentialité»
- **AOSTE** (CNRS, Université Nice Sophia Antipolis), modèles et méthodes pour l'analyse et l'optimisation des systèmes temps réel embarqués. Sous-branche «Systèmes embarqués et temps réel»
- **SECRET** (Inria), Sécurité, Cryptologie et Transmissions. Sous-branche «Algorithmique, calcul formel et cryptologie»

2.2.2 Chiffres clés

Effectif : **693** personnes, **599** scientifiques, **94** ingénieurs, techniciens et administratifs.

Recherche : **35** équipes de recherche, **1 à 2** nouvelles start-up créées chaque année, **280** contrats en cours, plus de **800** publications internationales à l'année.

2.3 Équipe Prosecco

L'équipe qui m'a accueillie est l'équipe *Prosecco* (dont le nom provient du vin blanc du même nom mais signifie aussi *Programming Securely with Cryptography*).

Elle a été fondée par *Karthikeyan Bhargavan*, qui dirige actuellement l'équipe [3].

2.3.1 Activités de recherche

Ses domaines de recherche se concentrent essentiellement sur les méthodes formelles et la sécurité logicielle et matérielle et sont les suivants :

- Vérification de protocoles (ProVerif, CryptoVerif, ProScript, F*) et de programmes
- Implémentations vérifiées de protocoles (miTLS), de primitives cryptographiques
- Attaques sur les protocoles (TLS, FlexTLS)
- Spécification de protocoles (TLS 1.3)
- Matériel sécurisé (Micro-Policies)
- Compilation sécurisée
- Test basé sur les propriétés (QuickChick, Luck)

3. http ://www.inria.fr/centre/paris/actualites/karthik-bhargavan-erc-consolidator-grants-2015

2.3.2 Composition de l'équipe

L'équipe est composée de 3 chercheurs permanents :

— **Karthikeyan Bhargavan** (enseignant à l'École Polytechnique, membre de Microsoft Resarch - Inria Joint Centre), travaillant sur les attaques, spécifications, implémentations concernant les protocoles et cryptographie.

— **Cătălin Hriţcu** (Membre du projet Micro-Policies), travaillant sur les méthodes formelles (vérification de programmes), sécurité matérielle, compilation sécurisée, test basé sur les propriétés.

— **Bruno Blanchet** (à l'origine de ProVerif et CryptoVerif), travaillant sur la vérification de protocoles (ProVerif, CryptoVerif).

D'une ingénieure de recherche, **Cécile**, travaillant sur la vérification de protocoles (ProVerif, CryptoVerif).

De 4 doctorants,

— **Nadim Kobeissi**, travaillant sur la vérification et l'implémentation vérifiée de protocoles. Il est à l'origine du logiciel *Cryptocat*.

— **Jean-Karim Zinzindohoué-Marsaudon**, travaillant sur l'implémentation vérifiée de primitives cryptographiques.

— **Benjamin Beurdouche**, travaillant sur la spécification, les attaques et l'implémentation liée aux protocoles.

— **Yannis Juglaret** (mon maître de stage) travaillant sur la sécurisée matérielle et la compilation sécurisée.

Au moment de mon arrivée, l'équipe avait déjà plusieurs stagiaires.

L'équipe est aussi à l'origine de nombreux outils : F* (Avec MSR), CryptoVerif, ProVerif, miTLS, FlexTLS, QuickChick.

2.4 Concurrence dans la recherche

La recherche est un milieu hautement compétitif :

— Il y a peu de postes de chercheur

— Il y a peu d'articles acceptés dans les plus grandes conférences et journaux

— Il faut avoir un article accepté à de grandes conférences afin d'obtenir un poste ou de monter les échelons quand on en a déjà un

De nombreux excellents articles sont rejetés par manque de place par exemple. Contrairement à la majorité des domaines scientifiques il faut savoir qu'une publication en conférence a plus de valeur qu'une publication dans un journal.

Dans certaines conférences il est très très difficile de voir son article être accepté (Oakland S&P, POPL). De plus il existe une course à la publication où d'autres chercheurs travaillant sur le même sujet peuvent publier avant et ainsi diminuer la valeur des travaux en cours mais pas encore proposés.

Quelques noms de grandes conférences en informatique liées à Prosecco :

Sécurité informatique : IEEE Symposium on Security and Privacy (Oakland S&P), IEEE European Symposium on Security and Privacy (Euro S&P), IEEE Symposium on Computer Security Foundations (CSF), USENIX Security Symposium (USENIX)

Langages de programmation : ACM Symposium on Principles of Programming languages (POPL)

Chapitre 3

Contexte du projet

3.1 L'assistant de preuves Coq

Coq est un **assistant de preuve** créé en 1984 par INRIA avec l'aide de divers contributeurs (ENS Lyon dans les années 90, École Polytechnique, CNRS, Université Paris Sud, Université Paris Diderot).

Un assistant de preuve est un outil logiciel qui permet d'exprimer des faits ou spécifications puis de construire des preuves formelles de certaines propriétés instruction par instruction. On **programme** en quelque sorte la preuve. L'assistant de preuve va ensuite **vérifier automatiquement** la validité de la preuve.

Avec Coq, on peut aussi programmer de façon classique mais sous un paradigme de **programmation fonctionnelle** (voir Glossaire) basé sur le langage *OCaml* avec certaines contraintes (Voir Annexe : Quelques fondements simples de Coq).

```
Require Import Induction.

Theorem plus_0_r :
  forall (n:nat), n + 0 = n.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0". reflexivity.
  Case "n = S n'". simpl. rewrite -> IHn'.
  reflexivity.
Qed.
```

```
2 subgoals
_____________________________________(1/2)
0 + 0 = 0
_____________________________________(2/2)
S n' + 0 = S n'
```

FIGURE 3.1 – *Preuve de "Pour tout n, n+0=n". On remarque qu'avec Coq on valide la cohérence de la preuve ligne par ligne. On peut apercevoir à droite les cas à prouver pour valider la preuve complète. Cette validation de preuves se fonde sur la correspondance de Curry-Howard (Voir Annexe : Quelques fondements simples de Coq).*

Il existe de nombreux assistants de preuves (HOL, Isabelle, Agda, PVS...) qui sont chacun basés sur des théories différentes mais Coq figure parmi les plus populaires.

Il est utilisé pour vérifier des propriétés sur des **programmes**. *CompCert* est un exemple de compilateur C formellement certifié. Mais aussi pour les **preuves mathématiques**. Le théorème de *Feit-Thompson* et le théorème des *4 couleurs* sont de grands théorèmes connus pour avoir été prouvés avec Coq.

Remarque : il peut-être intéressant de constater que Coq est français, qu'il est basé sur un système appelé *Calculus of Constructions (CoC)* fondé par *Thierry Coquand* et que Gallina signifie poule en latin.

3.2 Projet « Micro-Policies »

Le projet « Micro-Policies » est un projet de recherche initié par un article nommé «*Micro-Policies : Formally Verified, Tag-Based Security Monitors*» [4] dont les auteurs sont *Arthur Azevedo de Amorim et al.*

Cet article propose un mécanisme de sécurité, les **micro-politiques** qui sont des ensembles de règles de bas niveau directement intégrées dans un micro-contrôleur. À chaque instructions exécutées par un programme compilé, un système appelé *moniteur* va, de façon dynamique, vérifier s'il y a une violation des règles. Ce mécanisme repose sur un **système de tags** où chaque case de la mémoire et des registres sont associés à une étiquette qui porte des informations.

Tags	DATA	DATA	DATA	DATA	KEY k	DATA	SEALED k
Données	x_0	x_1	x_2	x_3	x_4	x_5	x_6

TABLE 3.1 – *Exemple de mémoire taguée où certains données sont verrouillées par une clé k contenue dans la mémoire sous le tag* KEY. *Le but sera d'empêcher la lecture des cases verrouillées si la bonne clé n'est pas fournie.*

Dans l'exemple ci-dessus, on peut faire appel à des "services de moniteur" qui permettent de gérer les tags. On aura par exemple un service *seal_addr* et *unseal_addr* pour, respectivement, verrouiller et déverrouiller une case de la mémoire selon les paramètres passés.

Voici un exemple de règle de micro-politique décrite symboliquement :

STORE := (•, DATA, DATA, t_{src}, –) → (•, t_{src}) où STORE est le nom d'une instruction du langage assembleur permettant de stocker une donnée dans une case mémoire d'adresse donnée en paramètre.

Cette règle énonce que si le registre r_{pc} (espace mémoire) contenant l'adresse de l'instruction à exécuter a le tag • (tag neutre constant), le tag de l'instruction courante est DATA, l'adresse où l'on veut stocker correspond à une case de tag DATA et que la donnée que l'on veut stocker est de type t_{src} alors on peut exécuter le stockage et on obtient le tag de r_{pc} et le nouveau tag de la case où l'on veut stocker qui est t_{src} (écrasement de tag).

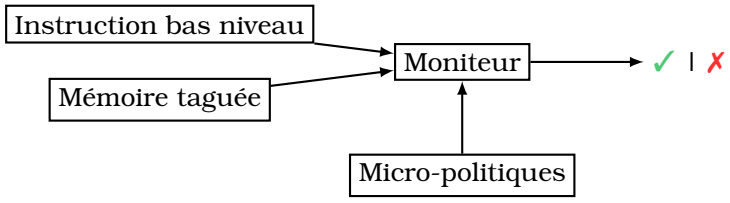

FIGURE 3.2 – *Représentation du fonctionnement des moniteurs. Si les entrées sont conformes, le moniteur valide l'action et autorise son exécution. En cas de violation des micro-politiques le moniteur empêche toute action supplémentaire.*

Cet article présente finalement comment raisonner formellement sur ces mécanismes et propose des exemples d'applications dont celui présenté dans le tableau ci-dessus mais aussi un système d'isolation de compartiments de mémoire au sein d'un programme.

3.3 Projet « Beyond Good and Evil »

« Beyond Good and Evil : Formalizing the Security Guarantees of Compartmentalizing Compilation » est le nom d'un article rédigé par *Yannis Juglaret, Cătălin Hriţcu et al*[1] lié à un exemple d'application de l'article « Micro-Policies » : la **compilation compartimentée sécurisée** (isolation de la mémoire d'un programme). C'est un article de sécurité informatique formelle.

3.3.1 Problématique

Certains langages comme le langage C, sont très répandus mais ne sont pas sûrs : ils présentent des **comportements indéfinis**.

Le langage C possède une norme qui spécifie le comportement que les programmes devraient avoir. Cependant certains cas ne sont pas pris en compte et la norme laisse la liberté aux compilateurs de gérer ces comportements indéfinis. Prenons un exemple :

...	0	1	2	-	...
...	Donnée	Donnée	Donnée	Code du programme	...

TABLE 3.2 – *On a un tableau de 3 cases allouées dans la mémoire. Que se passe-t-il si, dans une situation particulière, un attaquant provoque un accès à une case mémoire au delà du tableau ?*

On voit dans l'exemple ci-dessus que les comportements indéfinis rendent les programmes vulnérables et sensibles à la corruption de mémoire. Il pourrait par exemple être possible de lire ou d'écrire (Buffer Overflow) dans le code du programme afin de détourner son fonctionnement ou de récupérer des informations.

La raison pour laquelle ce type de comportement n'est pas anticipé est qu'une vérification dynamique ralentit l'exécution du langage et que dans certains cas ces comportements indéfinis sont nécessaires pour travailler dans un contexte de bas niveau [3]. Vitesse d'exécution et programmation bas niveau sont des caractéristiques essentielles du langage C.

Cependant certains chercheurs ont tenté de proposer des vérifications dynamiques (pendant l'exécution du programme) mais la vitesse d'exécution en est affectée [6].

3.3.2 Proposition

L'article se focalise sur un mécanisme de sécurité appelé **compartimentation** *(En anglais : compartmentalization)*. Pour limiter les exploitations de failles liées à la mémoire, l'idée est de découper un programme en plusieurs unités isolées avec des interactions limitées entre elle et une absence de confiance.

On divise un programme en unités appelées **composants** qui respectent la définition d'une **interface**. Dans le langage C on peut imaginer que les composants sont les fichiers sources (.c) et les interfaces les fichiers d'en-tête (.h).

1. J'ai aussi été ajouté comme auteur de l'article

On possède un langage **source** dont les programmes sont représentés par des expressions qui s'évaluent pas à pas. On représente un programme dans le langage **cible** (compilé) par une suite ordonnée d'instructions (langage assembleur). Le **compilateur** est représenté de façon abstraite par des règles de conversion entre le langage source et cible.

L'article propose une nouvelle propriété de sécurité appelée **Secure Compartmentalizing Compilation (SCC)** qui formalise les garanties formelles offertes par le système de compartimentation en proposant un modèle d'un programme, de son attaquant et de leurs interactions.

Pour modéliser un programme et son attaquant on découpe un programme complet en deux **programmes partiels** complémentaires. On peut ainsi voir cette modélisation comme un jeu avec un joueur et un opposant chacun possédant un certains nombre de composants et interagissant via des **actions**.

Dans notre système, ces actions ne pourront être que les suivantes : CALL l'appel de fonction, RETURN le retour de fonction et ✓ la terminaison de programme. On ne considère qu'un type particulier d'attaque où on ne retient que les interactions d'appels et de retours.

On ajoute aussi une propriété particulière au compilateur : avant de passer la main au joueur adverse on efface tous les registres (espaces mémoires nommés) en mettant leur valeur à 0 et chaque composant communiquent via le registre de communication r_{com}. Cela permet de s'assurer que l'attaquant ne se serve pas des stockages intermédiaires dans la mémoire afin de lire des données qui pourraient être sensibles par exemple.

Concernant les micro-politiques, leur rôle est d'apporter un environnement matériel concret et réel afin d'exploiter les mécanismes de sécurité proposés. On va pouvoir par exemple matérialiser le système de compartimentation avec des tags et restreindre l'accès aux composants non autorisés.

Finalement l'article donne une preuve de la propriété *SCC* qui énonce que pour tout scénario d'attaque (découpage programme-attaquant possible), les interactions entre les composants de l'attaquant et ceux du programme de bas niveau ne vont pas au delà de celles pour un contexte de haut niveau. Ainsi avec la propriété du compilateur, les attaques sur un programme compilé sont restreint à celles possibles dans le code source (spécification).

Il faut tenir compte du fait que cette propriété est une **garantie formelle de sécurité**, qui ne protège pas mais assure l'exclusion de certains comportements défavorables.

Chapitre 4

Rôle joué

4.1 Objectif du stage

Mon travail s'insère directement dans l'article « Beyond Good and Evil : Formalizing the Security Guarantees of Low-Level Compartmentalization » [3] et est considéré comme une contribution à la section 4 de l'article [1].

Afin de vérifier que *SCC* est vrai, les auteurs de l'article souhaitent **prouver des propriétés** à l'aide de l'assistant de preuve Coq. Dans la section 4, ils ont conçu un **langage très simple** représentant un langage impératif comme le C. Il est très basique et contient des opérations de base (conditions, récursion, appels de procédures, tableaux, accès et écriture dans les tableaux, opérations binaires basiques). On a ensuite un langage de bas niveau proche de l'assembleur et un compilateur.

Mon travail consistait à **définir** ces deux langages (source et cible) avec Coq (en utilisant l'aspect programmation fonctionnelle) et de **prouver** certaines propriétés qui étaient initialement considérées comme "hypothèses" afin de les passer à "lemme" ou "théorème" (en utilisant l'aspect assistant de preuve).

4.2 Organisation du travail

La première semaine a été dédiée à une **auto-formation** sur Coq par le livre numérique *Sofware Foundations* [1] (Trois des auteurs travaillent sur le projet Micro-Policies et sont auteurs de l'article *Beyond Good and Evil* !). Pendant cette même semaine et la suivante, j'ai commencé à **lire les articles** liés au projet [4], [3].

Yannis Juglaret m'a proposé, dès les premiers jours le planning suivant :

1. Formalisation des sémantiques
2. Preuves sur les sémantiques
3. Formalisation sur le compilateur
4. Preuves sur le compilateur
5. Formalisation du problème que le compilateur résout et la propriété associée

1. Au début de mon stage l'article était toujours en cours de rédaction.

6. Prouver que le compilateur satisfait la propriété associée

Mon maître de stage a été très disponible pendant chacune de ces phases (sauf dans certains cas comme la période de finalisation de l'article *Beyond Good and Evil*). Les échanges se faisaient par mail pour les questions courtes et par visite directement dans mon bureau quand des explications plus profondes étaient nécessaires.

4.3 Environnement de travail

Au niveau matériel, j'ai choisi de travailler sur mon propre ordinateur portable. Je disposais aussi d'un tableau blanc, de cahiers, feutres, stylos fournis par Inria.

Au niveau logiciel, voici une liste exhaustive des outils utilisés dans le projet :

— **Coq (CoqIDE)**, pour définir et prouver des propriétés.
— **Emacs**, un éditeur de texte utilisé notamment avec le module *Org* pour écrire des documentations simples.
— **Git**, un système de gestion de version qui permet de gérer un dépôt de fichiers pour le projet.
— **GForge**, une plateforme de gestion de projet qui permet de gérer une mailing-list et Git.

```
  ψ ⊢ φ well formed
————————————————————
φ whole well formed

** Big-step semantics...
** Small-step semantics...
** Examples : operational semantics reductions...
** Preservation and partial progress

*** Well-formedness invariants

η – component well-formedness invariants (preserved by reduction)
η.name
η.pnum
η.bnum
η.blens – list of lengths of the component's buffers

Γ – partial program well-formedness invariant
Γ ::= {η₀, ..., ηₙ}

Γ[C] – component access
Γ[C] ::= ηᵢ₀ when {ηᵢ | ηᵢ.name = C} = {ηᵢ₀}

wfinv(κ) ::= η where
  η.name = κ.name
  η.pnum = κ.pnum
```

FIGURE 4.1 – *Nous avions une sorte de documentation dans un fichier .org lisible avec Emacs. Cela permet de gérer des titres "déroulants" et d'écrire aisément des caractères spéciaux dans un fichier texte. Ce document décrit les détails des définitions liées au projet.*

Je vais maintenant décrire le travail que j'ai effectué en plusieurs parties. Il faut savoir que certaines parties n'ont pas été abordées par manque de temps et que beaucoup de choses ont été réalisées en parallèle.

Certaines choses ont **déjà été implémentées avec Coq** mais ne concernent pas la section 4 de l'article sur lequel je travaille.

Je vais commencer par présenter le langage source, cible puis définir un compilateur qui les relie.

Chapitre 5

Définition du langage source (2 semaines)

Nous avons donc un langage source présenté formellement dans la partie 4 de l'article «Beyond Good and Evil» [3].

Ce langage capture de façon stricte le comportement à risque d'un langage comme le C utilisant des tableaux (appelés *buffers* plus généralement) et des appels de fonctions. On s'abstrait de tout autre chose comme les variables, leur affectation ou les boucles dont le comportement ne nous intéresse pas (on se contente de la récursivité).

Un **programme** (noté ϕ) est constitué de **composants** (notés κ d'identifiant C) qui contiennent des **procédures** (notés P_i) et des **buffers** (notés b_i).

Programme ϕ							
*Composant κ_0							
*Buffer b				*Procédure P			
v_0	v_1	...	v_n	e_0	e_1	...	e_n

TABLE 5.1 – *Structure d'un programme ϕ. (*) signifie qu'il peut y avoir plusieurs instances de l'élément concerné.*

On peut assimiler cette structure à la structure de la *programmation orientée objet* où un composant représente une classe qui contient des méthodes (procédures), et des attributs (buffers).

Je vais décrire dans cette partie comment j'ai modélisé le langage source en me basant sur sa description formelle sur papier. Le code source étant (très) lourd, peu d'aperçus seront présentés.

5.1 Syntaxe

Une procédure est représentée par une expression qui s'évalue en un entier. On ne travaille d'ailleurs que sur des entiers pour simplifier.

Sur papier, la syntaxe du langage est décrite de la façon suivante par une **grammaire** BNF ou non-contextuelle : $e ::= i \mid e_1 \otimes e_2 \mid \mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid b[e] \mid b[e_1] := e_2 \mid C.P(e) \mid \mathit{exit}$

Cette grammaire permet de définir ce qu'est une expression du langage de façon **récursive** : une expression est soit un entier i, soit un opérateur binaire appliqué à deux expressions etc.

Notons que dans ce langage, tous les identifiants (pour les procédures, composants, buffers) sont représentés par des **entiers naturels** (entiers positifs).

Astuce. On utilise le symbole ";" comme opérateur binaire afin de séquencer les instructions. ";" évalue les deux instructions et retourne celle de droite.

Dans Coq, la syntaxe est définie ainsi :

```
Definition component_id := nat.
Definition procedure_id := nat.
Definition buffer_id := nat.

Inductive expr : Type :=
| EVal : nat -> expr
| EBinop : binop -> expr -> expr -> expr
| EIfThenElse : expr -> expr -> expr -> expr
| ELoad : buffer_id -> expr -> expr
| EStore : buffer_id -> expr -> expr -> expr
| ECall : component_id -> procedure_id -> expr -> expr
| EExit : expr.
```

Les identifiants sont des alias pour le type *nat* dénotant le type des entiers naturels $\mathbb{N}$. On définit le type *expr* des expressions où chaque type d'expression est représenté par une fonction renvoyant une expression. On remarque que les procédures ne prennent qu'un seul paramètre. Par convention, ce paramètre est stocké dans la première case du premier buffer du premier composant.

```
Definition buffer : Type := list nat.
Definition procedure : Type := expr.

Definition component : Type :=
  let name := nat in
  let bnum := nat in
  let buffers := list buffer in
  let pnum := nat in
  let export := nat in
  let procedures := list procedure in
  (name * bnum * buffers * pnum * export * procedures).

Definition program : Type := list component.
```

Les deux premières définitions sont intuitives. Par contre, un **composant** est représenté par une séquence contenant son identifiant, son nombre de buffers, ses buffers, son nombre de procédures, ses procédures et *export* est un nombre qui correspond à la limite entre procédures publiques et privées (comme en *POO*).

On dit qu'un respecte la définition d'une interface :

```
Definition interface : Type :=
  let name := nat in
  let export := nat in
  let import := list proc_call in
  (name * export * import).
```

L'**interface d'un composant** (noté $\imath$) est simplement son nom, son *export* et l'*import* est une liste d'appels de procédures de la forme (C.P) qui va correspondre à l'ensemble des appels externes dans ses procédures. Par extension l'**interface d'un programme** est l'ensemble des interfaces de ses composants (noté Ψ).

Concrètement, l'interface d'un programme peut être assimilé aux fichiers d'en-tête (.h) dans le langage C. Ces fichiers d'en-têtes contiennent la définition de toutes les fonctions

C et le fichier source (.c) qui l'importe doit respecter cette définition. C'est le même principe dans notre cas.

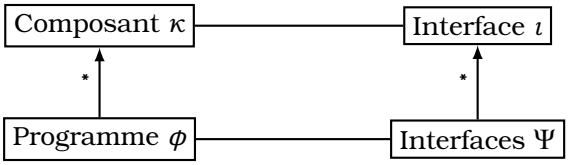

FIGURE 5.1 – *Représentation de la relation entre les notions définies précédemment. Un programme contient plusieurs composants et respecte l'interface de chaque composants.*

Dans Coq, il est aussi possible de définir des **notations**. Par exemple, le symbole (+) peut être vu comme une notation pour l'application de la fonction d'addition "plus x y". Voici la notation pour les conditions par exemple :

```
Notation "'IFB' e1 'THEN' e2 'ELSE' e3" :=
  (EIfThenElse e1 e2 e3) (at level 80, right associativity).
```

Listing 5.1 – On peut choisir la priorité et le sens de l'associativité.

5.2 Sémantique

Grâce à notre définition de la syntaxe nous pouvons maintenant écrire des programmes en assemblant des expressions.

Cependant, cette syntaxe n'est que texte dénué de sens. Il faut y ajouter une signification qu'on appelle **sémantique**.

Pour m'aider j'avais à ma disposition des exemples dans le livre numérique *Software Foundations* dont je me suis inspiré.

```
Definition eval_binop (e : binop * nat * nat) : expr :=
  match e with
  | (EEq, a, b) => if (beq_nat a b) then EVal 1 else EVal 0
  | (ELeq, a, b) => if (ble_nat a b) then EVal 1 else EVal 0
  | (ESeq, a, b) => EVal b
  | (EAdd, a, b) => EVal (a+b)
  | (EMinus, a, b) => EVal (a-b)
  | (EMul, a, b) => EVal (a*b)
  end.
```

Listing 5.2 – La fonction d'interprétation est une fonction qui, à partir d'un nom d'opération et de deux entiers, renvoie le résultat de l'opération correspondante. beq_nat correspond à (=) pour les entiers, ble_nat à (≤). ESeq est le séquençage d'instructions (; en C/C++/Java).

Il faut maintenant un moyen de «réduire» les expressions. La méthode que nous utilisons est appelée **sémantique opérationnelle** et est définie par un ensemble de **règles d'inférence**.

Une règle d'inférence est une règle formelle de la forme $\frac{A \quad B \quad \dots}{C}\ \mathcal{R}$ où $\mathcal{R}$ est le nom de la règle. Cette règle énonce que si les conditions A, B... (appelés prémisses) définies en haut sont satisfaites alors on peut en déduire C (appelé conclusion).

On distingue plusieurs méthodes d'évaluations par les règles d'inférences : l'évaluation *big-step* et l'évaluation *small-step*.

La première décrit une évaluation entière d'une expression à son résultat final et la seconde décrit une évaluation pas à pas. Nous utilisons l'évaluation **small-step**.

Une sémantique opérationnelle est donc un ensemble de règles d'inférences qui nous permettent d'évaluer un programme. Pour évaluer un programme, il suffit donc d'appliquer successivement des règles les règles définies. Pour comprendre la sémantique opérationnelle de notre langage, il faut introduire une nouvelle notion : les **configurations**.

Une configuration représente **l'état du programme** et est modélisée ainsi avec Coq :

```
1 Definition cfg : Type :=
2   component_id * state * call_stack * continuations * expr.
```

Listing 5.3 – Exemple de configuration. C'est une séquence de 5 éléments.

Une configuration est constituée des structures de données suivantes et permettent de modéliser l'état d'un programme à un point donné durant son exécution. L'exécution de programme est donc une transition d'un état à un autre à partir d'un état dit *initial*.

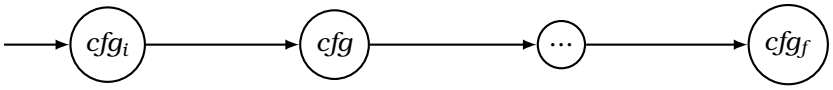

FIGURE 5.2 – *Représentation de l'exécution d'un programme à partir d'une configuration initiale jusqu'à une configuration finale (terminaison du programme)*

- **component_id (noté C)** : identifiant du composant courant.
- **state (noté s)** : un tableau à deux dimensions représentant une correspondance composant-buffer. Ainsi on peut avoir accès aux valeurs de chaque buffer de chaque composants. C'est un répertoire de valeurs.
- **call_stack (noté δ)** : pile d'appel. Permet d'empiler un historique d'appels de fonctions afin de revenir au point précédent après la terminaison de la procédure appelée.
- **continuations (noté K)** : une liste «d'expressions à trous» qui permet d'empiler les expressions à évaluer (une sorte de «TO-DO list» pour le programme).
- **expr (noté e)** : l'expression en cours d'évaluation.

Il faut clarifier cette histoire «d'expressions à trous» appelés **flat evaluation context**. Elles sont définies par la syntaxe formelle suivante :

$$E ::= \square \otimes e_2 \mid i_1 \otimes \square \mid \mathit{if}\ \square\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid b[\square] \mid b[\square] := e_2 \mid b[i] := \square \mid C.P(\square)$$

Une réduction d'une configuration à une autre est notée ainsi :

$\Delta \vdash cfg_1 \rightarrow cfg_2$ qui signifie que sous le **contexte** Δ, la configuration cfg_1 s'évalue en la configuration cfg_2. Un contexte est un tableau à deux dimensions simulant une correspondance composant-procédure afin de pouvoir récupérer le code de chaque procédures de chaque composants. Nous allons étudier 2 règles d'inférences :

$$\frac{}{\Delta \vdash (C, s, \delta, K, e_1 \otimes e_2) \rightarrow (C, s, \delta, \square \otimes e_2 :: K, e_1)} \text{ S\_BinOp\_Push}$$

Il n'y pas de conditions nécessaires. Cette règle signifie que : dans tous les cas, si on est en train d'évaluer une opération binaire, la configuration sera évaluée en une autre où le prochain objectif sera d'évaluer $(\square \otimes e_2)$ (La tâche est stockée à la tête de la configuration K) et où nous sommes en train d'évaluer le premier opérande (e_1) est l'expression courante. On aura une autre règle pour évaluer l'autre argument puis renvoyer le résultat de l'opération.

$$\frac{i \neq 0}{\Delta \vdash (C, s, \delta, \mathit{if}\ \square\ \mathit{then}\ e_1\ \mathit{else}\ e_2 :: K, i) \rightarrow (C, s, \delta, K, e_1)} \text{ S\_If\_Pop\_NZ}$$

Si on se trouve face à une configuration en train d'évaluer une condition, si cette condition i est évaluée comme étant différente de 0 (représentant FALSE dans notre langage d'entiers) alors la configuration s'évalue en une autre configuration où l'expression courante est e_1. C'est une règle assez intuitive.

J'ai dû transposer les règles formelles décrites sur papier dans Coq. J'ai dû aussi développer de nombreuses fonctions qui ne seront pas détaillées :

- Extraction des appels : fonction qui récupère tous les appels de fonctions dans une expression.
- Extraction des appels internes et externes : fonction qui récupère les appels internes et externes par rapport à un composant entré en paramètre.
- Fonction de mise à jour du contexte d'évaluation Δ et des buffers.
- Fonctions d'accès aux valeurs des buffers, aux procédures des composants.
- Génération de l'import pour les interfaces.
- Générations de la configuration initiale contenant le composant courant *Main* d'identifiant 0.
- Fonctions qui déterminent si des identifiants existent selon un certain contexte.
- D'autres fonctions secondaires

Dans Coq, les règles d'inférences sont modélisées de la façon suivante :

```
1  Reserved Notation "D ⊢ e '⇒' e'" (at level 40).
2  Inductive small_step (D: context) : cfg -> cfg -> Prop :=
3    (* S_BinOp_Push *)
4    | S_BinOp_Push : forall C s d K e1 e2 op,
5      D ⊢ (C, s, d, K, EBinop op e1 e2) ⇒ (C, s, d, (CHoleOp op e2)::K, e1)
6    (* S_If_Pop_NZ *)
7    | S_If_Pop_NZ : forall C s d K e1 e2 i,
8      ~(i=0) -> D ⊢ (C, s, d, (IFB □ THEN e1 ELSE e2)::K, EVal i) ⇒ (C, s, d, K, e1)
9    (* Other rules... *)
10   where "D ⊢ e '⇒' e'" := (small_step D e e').
```

Listing 5.4 – Modélisation sous Coq de la sémantique opérationnelle small-step du langage source.

La flèche -> représente **l'implication logique**. A -> B signifie : si A est vrai alors B est vrai. On peut aussi voir cette sémantique opérationnelle comme une **relation binaire** mettant en relation deux configurations sous un contexte Δ.

5.3 Vérifications

Afin de vérifier que mes définitions étaient correctes j'ai réalisé quelques tests. Cela a permis de détecter plusieurs erreurs (de frappe notamment).

J'ai défini un programme dans notre langage personnalisé qui permet de calculer une factorielle :

```
1 IFB (EBinop ELeq (LOAD 0<<EVal 0>>) (EVal 1)) THEN
2   EVal 1
3 ELSE
4   EBinop EMul
5     (CALL 1:::0 WITH (EBinop EMinus LOAD 0<<EVal 0>> (EVal 1))) (LOAD 0<<EVal 0>>)
```

Listing 5.5 – Expression calculant une factorielle en utilisant des notations personnalisées. Le paramètre est stocké dans le buffer d'identifiant 0.

On suppose que cette fonction est la 0-ième procédure du premier composant. L'appel récursif se fait donc via "CALL 1 : : :0". A partir de cette fonction j'ai prouvé par application successives de règles qu'une configuration calculant ce programme avec comme paramètre "2" retourne une configuration finale avec l'expression "Eval 2" comme résultat.

On peut ensuite définir des états, des contextes et vérifier que les fonctions de mise à jour et d'accès fonctionnent bien.

Chapitre 6

Preuves : Partial Type Safety (3 semaines)

Partial Type Safety est le nom de la première vraie propriété que j'ai dû prouver. Il s'agit d'une propriété provenant des systèmes de types qui permet de montrer qu'un système de type est cohérent. La propriété originale est *Type Safety* mais ici, à cause des comportements indéfinis on ne peut prouver qu'une version partielle. Comme nous travaillons sans types (juste des entiers) notre notion de typage correspond à la «bonne-formation» d'une expression.

Je présente dans cette partie ce qui définit qu'un élément du langage source est « bien formé » puis énonce la propriété sous forme d'un théorème composé de deux lemmes : *partial progress* et *preservation*.

6.1 Règles de « formation correcte » ou Well-formedness

Pendant mon stage je dispose de nombreuses règles d'inférences qui permettent de déduire pour chaque élément du langage (programme, composant, interface, continuations, pile d'appel, expression, configurations...) s'il est **bien formé**.

On introduit encore un nouveau concept : les **invariants**.

Un **invariant de composant** (noté η) contient des informations statiques et fixes concernant un composant comme son identifiant, son nombre de buffers, son nombre de procédures et un tableau qui contient la taille de chacun de ses buffers.

Un **invariant de programme** (noté Γ) est simplement l'ensemble des invariants de composants correspondants à chaque composants d'un programme.

Comme il y a beaucoup de règles et qu'elles sont assez complexes je n'en présente que deux :

$$\frac{\Gamma \vdash s \text{ well formed} \quad \Psi;\Gamma \vdash \delta \text{ well formed} \quad \Psi[C];\Gamma[C] \vdash K \text{ well formed} \quad \Psi[C];\Gamma[C] \vdash e \text{ well formed}}{\Psi;\Gamma \vdash (C, s, \delta, K, e) \text{ well formed}} \text{ WF\_CFG}$$

En lisant la règle de bas en haut, une configuration est bien formée par rapport à un ensemble d'interfaces et d'invariants de composants si chacune de ses composantes est bien formée. La définition se fait en cascade.

$$\frac{extprocsin(\iota.name, e) \subseteq \iota.import \quad \forall (C.P \in intprocsin(\iota.name, e)).P \in [0, \eta.pnum) \quad bufsin(e) \subseteq [0, \eta.bnum)}{\iota;\eta \vdash e \text{ well formed}} \text{ WF\_EXPR}$$

Toujours en lisant de bas en haut, Une expression est bien formée par rapport à une interface et à un invariant de composant si :

— L'ensemble des appels de procédures externes dans l'expression sont définies dans l'import de l'interface (l'import correspond aux appels externes dans les procédures du composant associé).
— Tous les appels externes (indicés par un entier) font appel à un identifiant compris entre 0 et le nombre de procédures du composant courant (défini dans l'invariant de composant).
— Tous les buffers utilisés dans l'expression sont définis.

Problème rencontré : les définitions sur papier ignore tous les détails techniques et parfois cela passe à travers notre vigilance. La règle WF_CFG aux composants C de Ψ et Γ. Mais que se passe t-il s'ils n'existent pas ? Il faut ajouter une prémisse qui permet d'utiliser la règle d'inférence seulement si C est défini dans Ψ et Γ.

6.2 Lemme : Partial Progress

Si notre sémantique possède la propriété *Progress* alors cela signifie que toute configuration bien formée est soit une configuration finale soit elle s'évalue en une autre configuration.

La propriété *Partial Progress* est une variante qui prend en compte les comportements indéfinis : une configuration est soit finale, indéfinie ou s'évalue.

On dit qu'une configuration est **indéfinie** si elle de la forme (C, s, d, b[□] : :K, i) ou (C, s, d, (b[i] := □) : :K, i') et que i n'est pas un indice correct dans le buffer b (hors bornes). Formellement la propriété est décrite ainsi :

Lemme : $\forall \phi.\ \phi$ well-formed $\Rightarrow$
$\psi = interfaceof(\phi) \ \wedge \ \Gamma = wfinc(\phi) \ \wedge \ \Delta = procbodies(\phi) \Rightarrow$
$\forall cfg.\ \psi; \Gamma \vdash cfg\ well-formed \Rightarrow$
$cfg\ final \vee cfg\ undefined \vee (\exists cfg'.\ \Delta \vdash cfg \rightarrow cfg')$

Qu'on lit comme ceci : pour tout (∀) programme bien formé, sachant que Γ est son interface, Ψ son invariant et Δ les procédures de ses composants, pour toute configuration bien formée, soit elle est finale, soit elle est indéfinie soit il existe (∃) une autre configuration vers laquelle elle s'évalue.

6.2.1 Comment prouver

Pour prouver une implication de la forme $A \Rightarrow B$ il faut supposer A et prouver B. Quand on veut prouver qu'une propriété est vraie pour tout objet, un prend un objet quelconque puis on démontre que la propriété est vraie pour cet objet.

6.2.2 Preuve

On suppose qu'on a un programme ϕ quelconque bien formé et une configuration cfg quelconque bien formée. Prouvons que cfg est soit finale, indéfinie ou s'évalue.

Pour cela on décompose *cfg* en (C, s, σ, K, e) pour exhiber ses composantes avec la tactique **destruct** de Coq et on utilise la tactique **inversion** sur l'hypothèse que *cfg* est bien formé. On sait que *cfg* est bien formé, donc les formes possibles qu'elle peut prendre sont limitées. La tactique va générer autant de sous-objectifs à prouver que de formes possibles. En développant chaque expressions jusqu'au bout on arrive aisément à plusieurs centaines de cas. Il est impossible de traiter tous ces cas un par un. Nous devons utiliser des méthodes d'automatisation.

Pour cela Coq fournit un moyen d'appliquer des instructions sur tous les cas et une tactique **try (**x_1**,** x_2**, ...,** x_n**)** qui permet d'essayer d'exécuter un ensemble d'instructions, si une échoue, l'ensemble échoue.

- **Cas final** : Dans certains cas, la configuration aura la forme d'une configuration finale, on résout ces cas en essayant de dire que la configuration est finale. Cela va éliminer de nombreux cas.
- **Cas indéfini** : On fait la même chose pour les configurations susceptibles d'être indéfinies (accès buffer ou stockage buffer avec un indice *i*). Il faudrait ensuite considérer deux cas : soit la configuration est définie soit elle ne l'est pas. Dans le premier cas elle s'évalue (on l'éliminera dans le cas suivant). Dans le second cas on dit qu'elle est indéfinie ce qui correspond à la propriété que l'on veut montrer.
- **Cas évaluable** : Le cas où la configuration s'évalue est un peu plus subtile. Il faut exhiber une configuration cfg' tel que *cfg* s'évalue en cfg'. Évidemment on ne peut pas traiter les cas uns par uns. L'astuce est de développer une fonction récursive *eval_cfg*() d'évaluation automatique et calculatoire. On prouve tous les cas restants en disant que $cfg' = eval\_cfg(cfg)$.
- Tous les cas ont été éliminés car chaque configuration bien formée possible est soit finale, indéfinie ou évaluable. CQFD.

6.3 Lemme : Preservation

Si notre sémantique possède la propriété *Preservation*, alors pour tout programme bien formé et pour toute configuration bien formée, si elle s'évalue alors elle produit une configuration bien formée. Ainsi on préserve la notion de bonne-formation.

Lemme : $\forall \phi.\ \phi$ well-formed.
$\psi = interfaceof(\phi) \ \wedge \ \Gamma = wfinc(\phi) \ \wedge \ \Delta = procbodies(\phi) \Rightarrow$
$\forall cfg\ cfg'.\ (\psi; \Gamma \vdash cfg\ well-formed) \Rightarrow (\Delta \vdash cfg \rightarrow cfg') \Rightarrow (\psi; \Gamma \vdash cfg'\ well-formed).$

6.3.1 Preuve

On a comme hypothèse que ϕ est bien formé, qu'on a un *cfg* quelconque bien formé et que *cfg* s'évalue en cfg'. Montrons que cfg' est bien formé. Je ne détaille pas la preuve (qui fait 277 lignes). Pour prouver cette propriété :

J'ai déconstruit *cfg* en (C, s, σ, K, e). Ensuite j'ai utilisé **inversion** sur l'hypothèse de *cfg* s'évalue en cfg'. Coq va générer toutes les combinaisons possibles d'évaluation et nous devons prouver la propriété pour chaque cas. Pour prouver qu'une configuration est bien

formée (Règle d'inférence *WF_CFG* il faut prouver que ses composantes s, σ, K et e sont bien formées et que C est bien défini (dans les interfaces Ψ du programme ϕ). Ce qui multiplie le nombre de cas par 5. Encore une fois on se retrouve avec de très nombreux cas. Il faut tenter des choses sur chaque branches en parallèle quitte à échouer sur certaines. Voici quelques cas que j'ai dû prouver :

- Si l'état s ne change pas pendant l'évaluation. On sait que cfg s'évalue en cfg' et que cfg est bien formé donc son état est bien formé. Par conséquent l'état s de cfg' est bien formé.
- Même chose pour σ et K.
- Si on reste sur le même composant C après l'évaluation, on sait qu'il était défini dans cfg vu que cfg est bien formé. Donc C est aussi défini pour cfg'.
- cfg contient l'affectation de buffer b[i] := □ et s'évalue en cfg' où la case i du buffer b est mise à jour. De plus on sait que i est bien défini (sinon l'évaluation ne serait pas possible). On montre par plusieurs lemmes que une mise à jour conserve la bonne-formation.
- Il faut montrer que le corps d'une procédure suite à un appel est bien formé. Pour cela on considère deux cas. Soit l'appel est interne (appel une autre procédure du même composant), soit il est externe. Il faut ensuite utiliser inversion et de très nombreux lemmes pour déduire ce que l'on veut (c'est le cas le plus long à prouver).
- ...

De très nombreux lemmes ont été utilisés pour cette propriété et j'ai dû modifier après confirmation de mon maître de stage, certaines règles de well-formedness pour faciliter le raisonnement mais il reste à prouver qu'elles sont équivalentes aux règles originales (elles ne sont équivalentes que par intuition mais pas formellement).

6.4 Théorème : Partial Type Safety

Cette propriété est la conjonction des deux précédentes.

En dehors de ces propriété j'ai aussi tenté de prouver qu'à partir d'un programme ϕ bien formé, sa configuration initiale (générée automatiquement par une fonction) est aussi bien formée. Pour cela on a une configuration initiale :

$(main, s[main, 0, 0 \mapsto 0], [], [], \phi[main].procs[0])$ et on prouve que chaque composante est bien formée sachant que le programme l'est.

Pour cela on va notamment montrer que tout programme bien formé contient bien un composant *main* par exemple. Je n'ai pas complètement prouvé cette propriété, il me reste un sous-cas que j'ai délaissé pour passer à plus important (notons que cette propriété est déjà prouvée sur papier et pourtant des blocages sont présents sous Coq).

De plus j'ai aussi prouvé d'autres propriétés dont je ne parlerais pas : strong progress pour la sémantique opérationnelle sans notion de typage, déterminisme, équivalence forme normale/configuration finale, équivalence sémantique opérationnelle/évaluation calculatoire.

Chapitre 7

Définition du langage cible (Environ 1 semaine)

Le langage cible correspond au langage source après compilation. Dans cette partie, les cours de *Système* à l'IUT de Montreuil m'ont particulièrement aidé à avoir une intuition de comment les choses marchaient.

Il est assez dur de déterminer combien de temps sa définition m'a pris sachant qu'elle a été faite en parallèle avec le langage source et le compilateur selon mes moments de blocage.

Le langage cible ou langage de **bas niveau** entre en interaction avec le fonctionnement concret d'une machine. Nous allons explorer comment l'environnement du langage de bas niveau (mémoire, registres, instructions...) est organisé.

7.1 Organisation des données

Le langage cible interagit avec une **mémoire** que l'on peut voir comme un tableau d'entiers où chaque case est indicée par un autre entier appelé **adresse**. La mémoire va notamment contenir des instructions correspondant à des programmes à exécuter, les buffers correspondants, les procédures etc.

Un autre espace de stockage existe : les **registres**. Les registres sont des espaces de stockage dédiés et nommés. Ils permettent avoir un accès rapide à certaines valeurs qui doivent être stockées pour une courte durée. On y stocke notamment le pointeur d'instruction appelé r_{pc} *(program counter)* qui va contenir l'adresse de l'instruction en cours d'exécution.

```
1 Definition memory : Type := list nat.
2 Definition register : Type := nat.
3 Definition nb_regs : nat := 7.
4 Definition r_pc : register := 0.
5 Definition r_one : register := 1.
6 Definition r_com : register := 2.
7 Definition r_aux1 : register := ... (* liste non exhaustive *)
```

Listing 7.1 – L'identifiant des registres correspond à une adresse dans la liste de tous les registres. L'ensemble des registres constitue aussi une mémoire.

La mémoire contient des instructions mais pourtant il n'y a que des entiers, comment est-ce possible ? Il faut imaginer des fonctions de **codage** et de **décodage** qui mettent en relation un entier et une instruction et ses paramètres (notion d'*opcode* en assembleur).

7.2 Instructions et codes

On ne parle plus d'expression mais de **code** qui est une séquence d'instructions. Les instructions sont les suivantes :

- *Nop* : instruction sans effet *(No operation)*.
- *Const* $i \rightarrow r$: place un entier i dans le registre r.
- *Mov* $r_1 \rightarrow r_2$: déplace le contenu d'un registre dans un autre registre.
- *BinOp* $r_1 \otimes r_2 \rightarrow r_3$: réalise l'opération $r_1 \otimes r_2$ et place le contenu dans le registre r_3.
- *Load* $* r_1 \rightarrow r_2$: met dans le registre r_2 la valeur dans la mémoire à l'adresse contenue dans le registre r_1.
- *Store* $* r_1 \leftarrow r_2$: met le contenu du registre r_2 à la case mémoire de l'adresse contenue dans le registre r_1.
- *Jal* r : correspond à un appel externe. Permet de sauter à l'instruction à l'adresse contenue dans r puis stocke le point de retour (adresse courante+1) dans le registre r_{ra}.
- *Jump* r : permet de sauter à l'instruction d'adresse contenue dans le registre r.
- *Call C P* : correspond à un appel externe. On dispose d'une liste de points d'entrées qui répertorie l'adresse de chaque procédure de chaque composant. L'instruction saute au point d'entrée correspond à C et P et répertorie l'appel dans une pile d'appel (comme dans le langage source).
- *Return* : saute à la case mémoire stockée au sommet de la pile d'appel (retour d'un appel).
- *Bnz r i (Branch if not zero)* : Si le registre r contient une valeur différente de 0 alors on saute à i instructions plus loin.
- *Halt* : arrête l'exécution du programme.

Notons que les instructions ne sont pas exécutées de façon récursive comme dans le langage source mais de façon linéaire (instruction par instruction).

7.3 Sémantique

Définissons à présent la sémantique opérationnelle [1] du langage de bas niveau.

Voici comment sont définis les programmes :

```
1 Definition program : Type :=
2   (partial_program_interfaces * global_memory * entry_points).
```

Un programme bas niveau noté (ψ, mem, E), est composé de trois éléments. L'**interface de programme** est la même que dans le langage source sauf qu'elle est partielle. Cela signifie qu'elle peut contenir des cases vides.

La **mémoire globale** est un tableau à deux dimensions associant composants et mémoire. Chaque composants a donc sa propre mémoire. Elle est aussi partielle.

On a finalement les **points d'entrée** qui est un tableau à deux dimensions associant à chaque composants une liste de points d'entrée (adresse de début de chaque procédures).

1. On parle aussi de machine abstraite.

On retrouve aussi une nouvelle définition des configurations :

```
Definition state : Type := (component_id * protected_callstack * global_memory *
    registers * address).
```

Une configuration contient l'identifiant du composant courant, une pile d'appel, une mémoire globale, un ensemble de registres et l'adresse courante en lecture *(program counter)*.

Toutes les règles ont la même forme et leur sens est très proche de celui de la sémantique opérationnelle du langage source. Étudions une de ces règles.

$$\frac{mem[C, pc] = i \quad decode\ i = Jal\ r \quad reg[r] = i' \quad reg' = reg[r_{ra} \rightarrow pc + 1]}{\psi; E \vdash (C, \sigma, mem, reg, pc) \rightarrow (C, \sigma, mem, reg', i')} \text{ S\_Jal}$$

En lisant de haut en bas et de gauche à droite : si on accède à la mémoire du composant C à la case pointée par le pointeur pc et qu'en décodant son instruction on trouve $Jal\ r$, si la valeur du registre r est i' et qu'en mettant à jour la liste des registres avec r_{ra} contenant le point de retour on obtient reg' alors la configuration courante s'évalue en une autre configuration avec la nouvelle liste de registres et où le lecteur d'instructions est à l'adresse i'.

7.4 Autre travail effectué

En dehors de la définition du langage cible, j'ai prouvé que sa sémantique était déterministe. C'est à dire que chaque configuration ne possède qu'une seule réduction possible. La preuve est très similaire au déterminisme de la sémantique opérationnelle du langage source : on suppose que pour tout contexte, si on a une configuration cfg qui s'évalue en cfg_1 et cfg s'évalue aussi en cfg_2 alors on a forcément $cfg_1 = cfg_2$. Ainsi il n'existe qu'une seule réduction possible.

Il est maintenant temps de définir un compilateur afin de relier notre langage source et notre langage cible.

Chapitre 8

Définition du compilateur (Environ 1 semaine)

Le compilateur est un mécanisme permettant de transformer un langage en un autre exécutable par une machine.

La définition du compilateur était assez linéaire et sans grands soucis. Une fois de plus aucune preuve n'a été complétée.

8.1 Méthode de compilation

Il y a plusieurs choses que l'on doit compiler : les composants, les procédures, les expressions et les **programmes partiels**.

On considère une définition supplémentaire pour les programmes partiels. Un programme partiel est un programme (ensemble de composants) qui peut avoir des emplacements vides. Notre compilateur nous permet donc de compiler un programme incomplet.

La compilation d'une structure Λ est notée $(\Lambda \downarrow)$.

8.1.1 Compilation des expressions

La compilation d'une expression va générer une suite d'instructions réalisant une opération équivalente. Exemple :

$(EBinop\ +\ e1\ e2)\downarrow = (k\ e1)\downarrow +\!+ (PUSH\ r_{com}) +\!+ (k\ e2)\downarrow +\!+ (POP\ r_{aux1}) +\!+ [BinOp\ +\ r_{aux1}\ r_{com}\ r_{com}]$

Où (++) est l'opérateur de concaténation et *PUSH* et *POP* des **macros** c'est à dire des alias pour un ensemble d'instructions.

Cette règle de compilation énonce que la compilation d'une addition compile le premier opérande, le met dans le registre r_{com}, le second argument dans r_{aux1} puis génère une instruction additionnant le contenu de r_{aux1} et r_{com} et stockant le résultat dans r_{com}.

8.1.2 Compilation des procédures

```
1 Definition compile_proc (k:component) (P:procedure_id) : code :=
2   let lmain := 3 in
3   let lret := 3 in
4   (* lext *)
5   [Const 1 r_aux2] ++
6   (RESTOREENV k) ++
7   [Bnz r_one lmain] ++
8   (* lint *)
```

```
9    [Const 0 r_aux2] ++
10   (PUSH r_ra) ++
11   (* lmain *)
12   (STOREARG k r_com r_aux1) ++
13   (compile_expr k (nth P (get_procs k) EExit)) ++
14   [Bnz r_aux2 lret] ++
15   (POP r_ra) ++
16   [Jump r_ra] ++
17   (* lret *)
18   (STOREENV k r_aux1) ++
19   (CLEARREG) ++
20   [Return].
```

La compilation d'une procédure produit la compilation du code de la procédure entouré de quelques opérations additives. Si on provient d'un appel externe alors on va commencer la lecture de la procédure à partir de l'étiquette *lint* sinon *lext*.

Il y a ensuite tout une gestion des paramètres, sauvegarde et restauration de l'environnement et le nettoyage de registre (CLEARREG) qui est une propriété importante du compilateur.

8.1.3 Compilation des composants

La compilation d'un composant produit un programme de bas niveau. Soit κ un composant.

$(\kappa \downarrow)$ est (Ψ, mem, E) où

— Ψ est l'interface de κ.

— mem est la concaténation de tous les buffers de κ à la suite, suivi de la compilation de chacun de ses procédures, une case mémoire libre pour restaurer la valeur du pointeur de pile d'appel en changeant de composant puis un nombre fini de 0.

— E est la liste des points d'entrées de chaque procédures de κ.

8.1.4 Compilation d'un programme partiel

La compilation $(\phi \downarrow)$ d'un programme ϕ est simplement la fusion de la compilation de chacun de ses composants.

8.2 Propriétés de programme et preuves

On a plusieurs propriétés qui seront utiles pour la suite disant qu'un programme de haut ou bas niveau **termine** ou qu'il **diverge** (boucle à l'infini).

Concernant les preuves je les ai juste énoncées :

— **Compilation des programmes complets** : si un programme P est bien défini, si il termine alors sa compilation $(P \downarrow)$ termine, s'il diverge, sa compilation diverge.

— **Compilation séparée** : Si on a un programme partiel A et P bien définis, si la fusion de A et P notée $A[P]$ termine alors sa compilation séparée $A \downarrow [P \downarrow]$ termine. Il en est de même pour la divergence.

Maintenant que tous les éléments qu'il nous faut sont là, comment raisonner dessus et prouver des choses en reliant chaque notions que l'on a défini ? Il faut pour cela utiliser un formalisme appelé **sémantique de traces**.

Chapitre 9

Sémantique de traces (Environ 1 semaine)

Nous avons un langage source, un langage cible et un compilateur afin de passer de l'un à l'autre. Il faut définir ce qu'on appelle une sémantique de trace afin de raisonner sur ces éléments.

Un **système de transition** est un système où l'on a des états et que l'on transite entre eux. Une **sémantique de traces** pour un système de transition est un formalisme permettant de suivre les transitions.

Avant de définir à quoi une sémantique de trace pourrait nous servir je vais introduire un nouveau point de vue : une attaque est un **jeu** d'opposition entre un programme et son attaquant où chacun a le droit à des actions à chaque tours.

9.1 Modèle programme-attaquant

Définissons les joueurs de notre jeu. Un des objectifs est de modéliser formellement la notion de programme et d'attaquant. Pour cela on divise un programme complet en deux programmes partiels complémentaires. L'un est appelé programme (noté P ou Q) et l'autre **contexte** ou **attaquant** (noté A).

Ainsi toujours avec la vision d'un jeu d'opposition, chacun "contrôle" des composants et des **interactions** se produisent. Les interactions possibles sont l'appel de procédure dans un composants (interne ou externe), le retour de procédure ou la terminaison de programme.

En fait, on représente programme et attaquant de façon duale sans distinction entre le "bien" et le "mal" afin de représenter toutes les interactions possibles entre deux parties complémentaires d'un programme, d'où le titre "Beyond Good and Evil".

Là où intervient la sémantique de traces c'est qu'elle nous permet de modéliser les interactions entre un programme et son contexte comme une **trace** qui va être une liste d'actions qui peut représenter un **historique de jeu**. Cela va nous permettre de raisonner sur le comportement des programmes.

Déroulement d'une partie : celui qui possède le composant *main* commence la partie. Chaque joueur peut faire autant d'action internes qu'il veut (appel de procédure qu'on

contrôle). Lors d'une action externe (appel de procédure contrôlé par l'adversaire), on passe la main à l'adversaire et c'est à son tour de lancer une action.

Objectif d'une partie : L'objectif est assez subtil. Le but de l'attaquant est de provoquer un comportement différent s'il fusionne avec un programme partiel quelconque P puis avec une **variante** de P appelée Q, c'est à dire P où l'on a substitué le code par un autre toujours en respectant les interfaces. Ainsi il aura montrer que dans un certains scénario il provoque une anomalie de comportement qui n'est pas présente dans les autres. Les comportements possibles sont la terminaison et la divergence (programme qui boucle à l'infini).

Fin de partie : il y a deux moyens de terminer une partie, soit le programme provoque sa terminaison soit l'un des joueurs provoque une divergence.

A	κ_0		κ_2	κ_3	
P		κ_1			κ_4

TABLE 9.1 – *Exemple de découpage d'un programme complet en programme P et contexte A. Le contexte possède 3 composants et le programme 2.*

9.2 Définition des actions et des traces

Une action a est soit une action interne Ia ou externe Ea étiquetée par son **origine** (programme ou contexte).

Une action **externe** notée γ est soit un appel externe $C.P$, soit un retour de fonction $Return$ soit une terminaison de programme $\checkmark$.

Si elle vient du programme son suffixe est (!), si elle vient du contexte (?).

Une action **interne** est soit un appel interne $C.P$, un retour de fonction interne $Return$ ou une autre action τ (qui ne nous intéresse pas).

Si elle vient du programme son suffixe est (+), sinon (−).

Une **trace** notée t est une séquence finie d'actions. Les traces ne prennent en compte que les actions externes et comme chaque action externe passe le tour au "joueur suivant" on a une alternance programme-contexte.

Quelques exemples de traces :

- ϵ est la trace vide.
- $t.\gamma!$ est une trace quelconque qui se termine par une action externe provenant du programme.
- $t.\gamma!.\gamma?$ est une trace qui se termine par une action externe du programme puis du contexte.

Grâce à notre système on peut "pister" les actions du programme et de l'attaquant et chacun possède son historique d'actions de jeu.

Les traces possèdent ensuite de nombreuses autres définitions, comme leur propre sémantique de réduction (comme les sémantiques opérationnelles que l'on a défini, et des

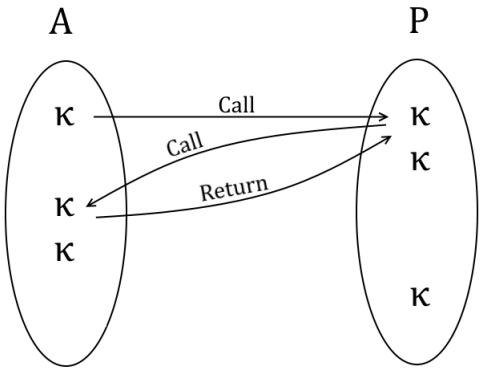

FIGURE 9.1 – *Une flèche représente une action et chaque chemin (partiel ou complet) est une trace.*

notions d'états, de piles d'appel et de bonne-formation des états qui ne seront pas définies ici car cela n'est pas nécessaire).

Je vais cependant définir quelques éléments nécessaires pour la suite.

9.3 Ensembles de traces

Forme d'un programme partiel (Shape)

Soit A et P deux programmes partiels complémentaires représentant un attaquant et un programme. On dit que chacun a une forme s.

La **forme** d'un des deux programmes partiels contient l'interface Ψ du programme complet $A[P]$ et l'ensemble des composants contrôlés par P noté $\{C_1, C_2, ..., C_n\}$.

On a les notations suivantes :

— $P \in^{\bullet} s$: le programme P a la forme s. C'est à dire que les composants qu'il contrôle sont ceux définis dans s.

— $A \in^{\circ} s$: le contexte A a la forme s. C'est à dire que les composants qu'il contrôle sont ceux définis dans Ψ en retirant les composants contrôlés par le programme $\{C_1, C_2, ..., C_n\}$.

On a les même notations pour les programmes de bas niveau mais on écrira plutôt a et p.

Définition des ensembles de traces

Les ensembles de traces sont définis pour les programmes de bas niveau seulement.

— $Traces_{\bullet s}(p)$: pour un programme p de forme s, une trace appartient aux traces p si elle correspond à l'historique de p à un tour quelconque.

— $Traces_{\circ s}(a)$: Symétriquement, on a la même notion pour un contexte de bas niveau a.

9.4 Canonisation des traces

Le compilateur que l'on a définit efface la valeur des registres sauf r_{com} suite à certaines actions comme les appels ou retours de fonction.

Cela permet de rendre la valeur des registres inaccessible à la fonction appelée vu qu'on ne lui fait pas confiance. On garde r_{com} car c'est le registre dédié à la communication inter-composants.

Cette propriété est en réalité essentielle car sans elle on ne pourrait pas prouver la propriété finale *SCC (Secure Compartmentalizing Compilation)*.

On définit une fonction connexe à ce nettoyage de registre appelé $\zeta\circ$ qu'on applique à une trace tel que pour une trace t, $\zeta \circ (t)$ produit la même trace t mais avec les registres (sauf r_{com}) mis à 0 après chaque action externe provenant du contexte.

Symétriquement on a la fonction $\zeta\bullet$ qui nettoie tous les registres sauf r_{com} pour les actions provenant du programme.

Chapitre 10

Compilation compartimentée sécurisée (3 semaines)

La compilation compartimentée sécurisée, abrégée SCC *(pour Secure Compartmentalizing Compilation)* est la propriété centrale que l'on cherche à prouver. Commençons par une définition informelle.

Secure Compartmentalizing Compilation : pour tout scénario d'attaque (partitionnement d'un programme en programme/contexte), les composants contrôlés par l'attaquant ne peuvent pas "causer plus de dégâts" aux composants contrôlés par le programme dans un environnement de bas niveau (post-compilation) qu'ils ne peuvent dans un environnement de haut niveau (langage source).

Il a été prouvé avec Coq par d'autres contributeurs du projet et dans l'article que cette propriété est en fait la conséquence de deux autres propriétés : la **correction de la compilation compartimentée** et **Structured Full Abstraction**.

Structured Full Abstraction était déjà prouvé dans l'article sous forme informelle, on m'a donc proposé de la transposer sous Coq pour la vérifier formellement.

10.1 Définition de Structured Full Abstraction

Pour deux programmes P et Q quelconques de forme $\in \bullet$ et tous deux fully-defined :
Théorème :
$(\forall A \in \circ s.\ A$ fully-defined par rapport à la forme $\bullet s \Rightarrow A[P] \sim_H A[Q])$
$\Rightarrow (\forall a \in \circ s.\ a[P\downarrow] \sim_L a[Q\downarrow])$

On dit qu'un contexte A est **fully-defined** ou *pleinement défini* par rapport à une forme $\bullet s$ si en fusionnant A avec un programme P de forme s on produit $A[P]$ et que sa configuration initiale $\mathcal{I}(A[P])$ n'aboutit à un comportement indéfini dans aucun cas. La configuration initiale étant juste une configuration générée automatiquement selon un programme et qui va contenir le composant *main* et la procédure *main*.

Le symbole $\sim_H$ signifie une équivalence de comportement de programmes de haut niveau (un programme se termine ou diverge, c'est à dire boucle à l'infini) et $\sim_L$ est

l'équivalent pour le bas niveau. Leur opposés (distinction de comportement) sont notés $\not\sim_H$ et $\not\sim_L$.

Cette définition signifie que si deux programmes P et une de ses variantes Q (C'est à dire P où l'on a remplacé le code orignal par un autre tout en respectant les interfaces) ont le même comportement lorsqu'ils sont mis en contact avec un contexte A quelconque de haut niveau alors ce comportement est préservé pour tout attaquant a de bas niveau.

Intuition : disons que P on cherche à savoir si un attaquant arrive à produire un comportement différent avec deux variations d'un même programme partiel. Si pour toutes ($\forall$) variations, le comportement ne change pas sous un contexte A, cela signifie qu'après compilation, aucun contexte a ne peut les distinguer et que par conséquent l'attaquant ne peut pas agir sur les éléments du code qui varient.

Clarifions cette histoire de distinction de comportement.

En pratique, on va plutôt utiliser la **contraposée** qui est plus facile à prouver et peut-être plus intuitive. Si l'on a une formule d'implication de la forme $(A \rightarrow B)$ alors sa contraposée est $(\neg B \rightarrow \neg A)$ où $\neg$ désigne la négation. Une implication et sa contraposée sont logiquement équivalents.

La variante basée sur la contraposée est décrite ainsi :

Pour deux programmes P et Q quelconques de forme $\bullet s$ et tous deux fully-defined :
Théorème : $\exists a.\ a[P\downarrow] \not\sim_L a[Q\downarrow]$
$\Rightarrow (\exists A.\ A$ fully-defined par rapport à la forme $\bullet s \wedge A[P] \not\sim_H A[Q])$

Cette formule signifie que s'il y a un attaquant qui provoque une distinction entre le comportement de deux programmes fully-defined alors il existe aussi un attaquant de haut niveau fully-defined qui les distingue.

La preuve étant assez complexe seule une intuition détaillée sera donnée afin de la comprendre (Elle utilise beaucoup de lemmes assez complexes pour la plupart).

10.2 Intuition de la preuve de Structured Full Abstraction

On procède avec une **preuve par construction** [1] à partir de la variante basée sur la contraposée de *Structured Full Abstraction*. Considérons qu'on a un attaquant a agissant sur le langage compilé et provoquant la distinction du comportement de deux programmes quelconques. On doit générer un attaquant A particulier full-defined tel qu'il distingue aussi les composants mais dans le haut niveau.

10.2.1 Algorithme de trace mapping / Propriété de définabilité

Grâce à la *correspondance de Curry-Howard* [2] nous pouvons voir les programmes comme des preuves. Afin de générer l'attaquant A que nous voulons on dispose d'un

1. Voir Annexe "Quelques fondements simples de Coq" - Logique intuitionniste.
2. Voir Annexe "Quelques fondements simples de Coq" - Correspondance de Curry-Howard

algorithme (développé par mon maître de stage avec le langage *OCaml*) qui est représenté avec Coq sous forme d'un théorème (plutôt hypothèse car cela n'a pas encore été prouvé).

Cet algorithme nous permet de générer l'attaquant sous certaines conditions :

1. On a une trace t tel que $t = \zeta \circ (t)$ (Ses registres sont déjà nettoyés)
2. Il existe un programme possédant une trace $(t.\gamma_1!)$ avec un γ_1 quelconque.

On va donc chercher à remplir ces conditions pour utiliser l'algorithme pour créer un attaquant de haut niveau A. Il restera à prouver que ce A distingue le comportement des programmes de haut niveau correspondant à ceux distingués par a.

10.2.2 Preuve informelle de Structured Full Abstraction

Soit a un attaquant de bas niveau, $(P \downarrow)$ et $(Q \downarrow)$ des programmes de haut niveau compilés. On sait que a distingue le comportement de $(P \downarrow)$ et $(Q \downarrow)$. Considérons, par exemple, que $a[P \downarrow]$ se termine et que $a[Q \downarrow]$ diverge (le cas inverse est symétrique et se prouve de la même façon).

Comme $a[P \downarrow]$ se termine, a et $P \downarrow$ partagent une trace t_i qui se termine par le symbole de terminaison ✓. On appelle t_p le plus long préfixe de t_i qui s'arrête au point où le comportement de $(Q \downarrow)$ change. Voici une représentation de ce qui se passe :

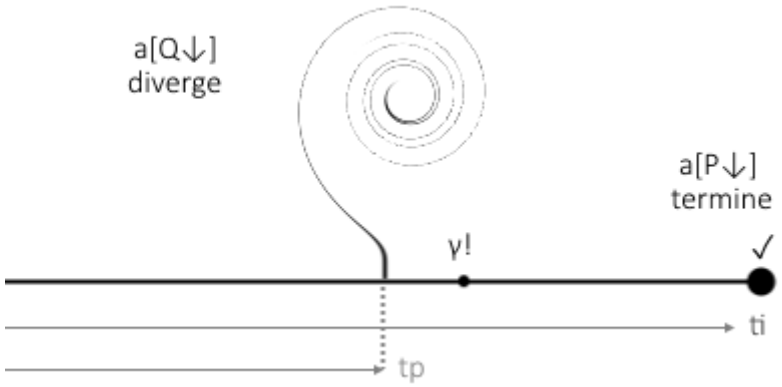

FIGURE 10.1

On considère que ce préfixe t_p est un *préfixe strict* c'est à dire qu'il est strictement plus petit que t_i. Comme il est strictement plus petit il existe une action qui le suit. On appelle cette action γ. Il a été prouvé dans l'article que cette action est forcément une action du programme (si c'est une action du contexte cela mène à une contradiction). On obtient donc la trace $tp.\gamma!$ appartenant aux traces de $(P \downarrow)$ (Elle ne peut pas appartenir à $Q \downarrow$ sans contredire notre hypothèse de plus long préfixe).

On note t_c la canonisation de t_p, c'est à dire que l'on a : $t_c = \zeta \circ (t_p)$. On sait que $tp.\gamma!$ est une trace de $(P \downarrow)$, un lemme nous permet de dire que sa canonisation est aussi une trace de $(P \downarrow)$. On sait maintenant que $\zeta \circ (tp.\gamma!)$ est une trace de $(P \downarrow)$.

On sait que la fonction $\zeta\circ$ n'affecte que les actions provenant du contexte. $\gamma!$ est une action du programme (car il possède le suffixe !) donc $\zeta \circ (tp.\gamma!) = \zeta \circ (tp).\gamma!$. Nous avons maintenant la trace qu'il nous fallait pour utiliser l'algorithme de trace mapping.

On utilise l'algorithme de trace mapping afin de générer un attaquant A qui distingue le comportement des programmes partiels P et Q. Il faut montrer que $A \downarrow [P \downarrow]$ termine et que $A \downarrow [Q \downarrow]$ diverge.

Pour prouver ces deux derniers faits on utilise la logique classique[3]. Cette logique nous permet de considérer un théorème appelé tiers exclu : peu importe P, soit P est vrai soit P est faux (plus formellement : $\forall P, P \vee \neg P$).

$A \downarrow [P \downarrow]$ termine : on considère que soit $\gamma_1! = \checkmark$ ou $\gamma_1! \neq \checkmark$ grâce au tiers exclu. Dans le premier cas la trace se termine par $\checkmark$, cela signifie que le programme provoque la terminaison du programme complet qui se termine donc. Dans le second cas, je ne donne pas de précision afin de rester informel mais l'algorithme de trace mapping nous permet de dire que si $\gamma_1! \neq \checkmark$ alors le contexte est obligé de réaliser l'action ($\checkmark$?) donc le programme complet termine.

$A \downarrow [Q \downarrow]$ diverge : on utilise successivement plusieurs instances du tiers exclu ce qui nous permet de former un arbre binaire de preuve où aux feuilles de l'arbre : soit on a ce que l'on veut c'est à dire $A \downarrow [Q \downarrow]$ diverge soit on a une contradiction, donc dans tous les cas "cohérents", $A \downarrow [Q \downarrow]$ diverge. Visualisons l'arbre de preuve :

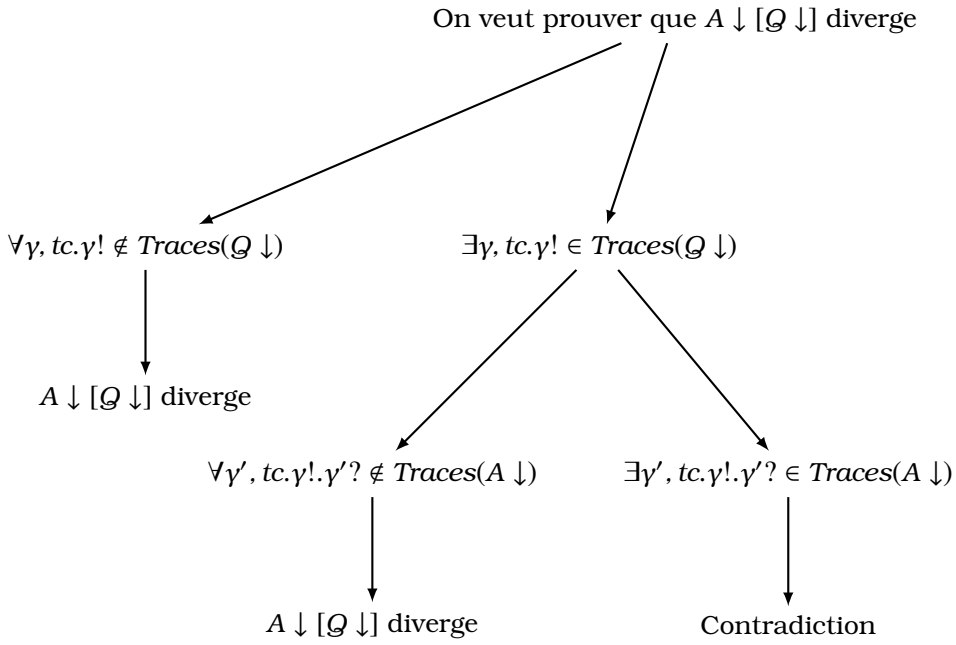

FIGURE 10.2 – *Arbre représentant la preuve de la divergence de $A \downarrow [Q \downarrow]$. Chaque embranchement représente une utilisation du tiers exclu.*

On commence tout en haut, à la racine de l'arbre. On ne connaît pas trop la situation de la partie donc on prouve la divergence ou la contradiction dans les cas possibles.

Commençons par remarquer que la négation de $\forall x, a \notin A$ est $\exists x, a \in A$ donc les deux cas sont bel et bien opposés. Dans la première utilisation du tiers exclu, soit le programme ne lance pas d'action $\gamma!$ soit il en lance une.

3. (Voir Annexe "Quelques fondements simples de Coq" - Logique intuitionniste)

— Dans le premier cas s'il n'y a pas d'action du programme qui suit cela veut forcément dire que $(Q\downarrow)$ bloque le programme en provoquant une divergence.
— Dans le second cas, soit le contexte ne lance aucune action supplémentaire soit il répond avec une autre action γ'?.
  — Dans le premier cas on a une divergence car il n'y a plus d'actions supplémentaires.
  — Dans le second cas, quelques lemmes nous permettent de montrer que ce cas est impossible et mène donc à une contradiction.

On sait maintenant que $A\downarrow[P\downarrow]$ termine et que $A\downarrow[Q\downarrow]$ diverge et par conséquent que $(A\downarrow)$ distingue le comportement de la compilation de P et Q dans le bas niveau (programme compilé). Il nous reste toujours à prouver $(\exists A.\ A$ fully-defined par rapport à la forme $\bullet s \wedge A[P] \not\sim_H A[Q])$ (voir énoncé du théorème).

Pour cela on énonce qu'il existe bien un A qui remplit ces conditions : on utilise le A que l'on a généré grâce à l'algorithme de trace-mapping.

Maintenant il nous reste à montrer que A provoque une différence dans le comportement de P et Q dans le haut niveau. Le lemme de **correction de compilation compartimentée** (Corollaire 4.3 de l'article [3]) nous permet de dire que la compilation préserve le comportement. Donc on sait que si les comportements sont différents dans le haut niveau, ils le sont aussi dans le haut niveau. On applique ce fait pour justifier que A distingue bel et bien le comportement de P et Q dans le haut niveau.

La propriété est maintenant prouvée.

□

Bilan

Bilan du travail

L'ensemble de mon travail prend la forme de fichiers sources Coq. Je propose dans cette section un commentaire concernant chaque fichier dont je suis l'auteur en exposant brièvement des avis et des difficultés.

- **Source.v** (2170 lignes) : C'est le langage source qui possède le plus de définitions. Mon principal problème a été d'adapter les définitions à l'aspect calculatoire de la programmation. Les preuves figurant dans ce fichier n'avaient pas été prouvées auparavant et ont nécessité la production de nombreux lemmes. Cela a été la partie la plus conséquente.
- **Target.v** (531 lignes) : Une fois que l'on a vu le langage source, le langage cible est plutôt simple mais cache de nombreuses subtilités à cause de sa proximité avec le fonctionnement de la machine : tout est représenté (programmes et données) dans la mémoire qui est un tableau d'entier comme dans une vraie machine.
- **Compiler.v** (356 lignes) : La définition du compilateur a été plutôt simple : c'est un ensemble de fonctions de conversations du langage source vers le langage cible. Sur papier, des fonctions étaient définies de façon circulaire. Cette circularité a dû être brisée par quelques astuces.
- **Shape.v** (220 lignes) : La définition des formes n'a pas posé de problème mais je me suis rendu compte qu'il fallait redéfinir beaucoup de choses dans le langage cible et source. Certaines définitions devaient être partielles : un programme peut contenir des composants ou des "trous". Autrement, cela aurait causé plus de problèmes.
- **TraceSemantics.v** (520 lignes) : La définition de la sémantique de traces n'a pas posé de réel problème mais a nécessité de longues explications de la part de mon maître de stage. J'ai personnellement trouvé que c'était une notion particulièrement élégante.
- **SCCProof.v** (1032 lignes+) : Malgré le fait qu'il suffisait de transposer la preuve de SCC sur papier avec Coq, elle cachait beaucoup de subtilités qu'on ne retrouve que dans un contexte technique. Une preuve sur papier omet ce qui est évident pour l'être humain mais ces choses ne le sont pas pour la machine. De nombreux lemmes ont dû être conçus et prouvés. J'ai notamment dû faire appel à la logique classique afin d'utiliser le tiers exclu et de faciliter la preuve.

Conclusion

Ce stage m'a apporté une vision concrète de l'environnement de la recherche scientifique : j'ai pu assister à quelques séminaires sur des thématiques actuelles de recherche (sécurité des protocoles, preuve de programme). J'ai pu assister à la soutenance de thèse de Jacques-Henri Jourdan de l'équipe *Gallium* sur l'analyse statique par interprétation abstraite. J'ai aussi pu observer le travail des chercheurs de mon équipe (rédaction de l'article *Beyond Good and Evil*, télé-conférences, organisation du travail) et assister à des événements d'Inria comme la «demi-heure de science» mensuelle.

D'un point de vue technique, ce stage m'a permis de m'isoler de la programmation classique vue à l'IUT (programmation impérative, orientée objet) en expérimentant la programmation fonctionnelle. J'ai aussi pu acquérir des connaissances par le biais de questions posées à mon maître de stage et ainsi découvrir de nouvelles branches de l'informatique. Initialement, les preuves mathématiques ne m'étaient pas vraiment familières. L'utilisation de Coq m'a permis de me familiariser un peu plus avec. C'est une activité essentielle dans la recherche scientifique.

De plus je pense avoir reçu un réel soutien de la part de mon maître de stage et de son superviseur de thèse pour d'éventuelles poursuites dans le domaine de la recherche. Il m'a été proposé d'intégrer mon nom à l'article *Beyond Good and Evil* et d'assister à la conférence *CSF 2016 (Computer Security Foundations)* au Portugal afin d'avoir une première vision réelle d'une conférence scientifique.

Par rapport aux enseignements de l'IUT, les cours de Système m'ont particulièrement aidé à avoir une intuition sur les notions de bas niveau (registres, mémoire, langage assembleur, architecture matérielle...).

Du point de vue du projet dans lequel je travaillais, mon travail a permit de fournir une modélisation des notions formelles du projet, de détecter des erreurs de spécification et d'apporter une force supplémentaire à l'article (destiné à être présenté à la conférence CSF2016) en prouvant formellement certaines propriétés pour ainsi les passer d'hypothèse à lemme ou théorème.

Épilogue

Après mon stage, sur le court terme, j'aimerais pouvoir continuer de maintenir ma contribution au projet afin de compléter ce que je n'ai pas pu faire.

Sur le long terme, j'aimerai continuer à approfondir mes connaissances sur Coq et éventuellement d'autres assistants de preuves. Enfin, j'aimerais pouvoir faire mes prochain stages dans d'autres organismes de recherche afin d'avoir des visions différentes (CNRS, Universités) voire découvrir un environnement plus industriel (entreprises privées).

Annexes

Annexe A

Exemple de preuve Coq

Afin de construire des preuves, j'utilise CoqIDE qui me fournit un environnement avec de nombreuses fonctionnalités et une interface graphique adaptée. Il est aussi possible d'utiliser Emacs et d'intégrer Coq, cela permet notamment d'écrire des caractères spéciaux (utile pour les notations personnalisées).

Je présente ici une preuve de $n + 0 = n$ qui est pourtant évident sur papier mais pas automatique. Avant tout il faut une modélisation des entiers naturels (c'est à dire les entiers positifs) afin de raisonner dessus.

Modélisation des entiers naturels

Coq fournit sa propre modélisation des entiers naturels avec le type NAT .

```
1 Inductive nat : Type :=
2 | O : nat
3 | S : nat -> nat.
```

Listing A.1 – Modélisation des entiers naturels

On définit les entiers de façon **inductive** comme étant une suite de successions du nombre 0.

Pour cela on définit un **type inductif** énonçant que 0 est de type NAT et que S est une fonction qui à un entier associe un objet de type NAT. Le nombre 3 sera par exemple représenté par (S (S (S 0))). Heureusement Coq possède son propre mécanisme de traduction automatique vers les chiffres arabes qui nous sont plus familiers.

Preuve

```
Require Import Induction.

Theorem plus_0_r :
  forall (n:nat), n + 0 = n.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0". reflexivity.
  Case "n = S n'". simpl. rewrite -> IHn'.
  reflexivity.
Qed.
```

```
2 subgoals
_____________________________________(1/2)
0 + 0 = 0
_____________________________________(2/2)
S n' + 0 = S n'
```

FIGURE A.1 – *Preuve de n+0=n qui est pourtant évident sur papier.*

— Au départ rien n'est surligné en vert. On commence par écrire l'**énoncé** du théorème qui correspond à une expression logique qui est soit **prouvable** soit **non prouvable**. Ici on dit simplement que pour tout n, $n + 0 = n$.

— On appuie sur les touches Ctrl+Bas (varie selon les versions) pour faire **vérifier l'énoncé** par Coq. S'il est bien exprimé il est surligné en vert puis on progresse ainsi.

— On **introduit** un n quelconque puis on cherche à prouver la propriété. On lance une **preuve par induction** sur n (preuve par récurrence) : on prouve la propriété pour $n = 0$ puis en supposant qu'elle est vraie pour un n précis on prouve qu'elle l'est aussi pour le n suivant (Sn). Cela permet de prouver que la propriété est vraie pour 0 et qu'elle se propage par successivement. Elle sera donc vraie pour tout entier.

— On valide les instructions et deux sous-preuves doivent être fournies pour compléter la preuve (qu'on voit à droite).

— Le **premier cas** est évident. On utilise comme argument la "réflexivité de l'égalité" qui nous permet de dire "D'après la définition de l'égalité, les deux côtés sont égaux". On valide avec Ctrl+Bas pour voir ce que Coq en pense. Il simplifie lui même l'expression et vérifie que l'on a bien la même chose des deux côtés puis surligne en vert.

```
Theorem plus_0_r : forall (n:nat), n + 0 = n.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0". reflexivity.
  Case "n = S n'". simpl. rewrite -> IHn'.
  reflexivity.
Qed.
```

```
1 subgoal
Case := "n = S n'" : String.string
n' : nat
IHn' : n' + 0 = n'
______________________________________(1/1)
S n' + 0 = S n'
```

— Dans le deuxième cas, une hypothèse nommée *IHn'* apparaît à droite. Et l'objectif a changé. L'addition (S n') + m est définie récursivement comme étant S (n' + m). On peut utiliser une simplification.

```
Theorem plus_0_r : forall (n:nat), n + 0 = n.
Proof.
  intros n. induction n as [| n'].
  Case "n = 0". reflexivity.
  Case "n = S n'". simpl. rewrite -> IHn'.
  reflexivity.
Qed.
```

```
1 subgoal
Case := "n = S n'" : String.string
n' : nat
IHn' : n' + 0 = n'
______________________________________(1/1)
S (n' + 0) = S n'
```

— On sait que $n' + 0 = n'$. On effectue une réécriture syntaxique vers la droite et on utilise la réflexivité de l'égalité. La propriété est prouvée quand l'instruction *Qed* est surlignée.

Annexe B

Quelques fondements simples de Coq

Je présente ici quelques intuitions sur le fonctionnement de Coq.

Programmation fonctionnelle

Coq permet de programmer avec un paradigme fonctionnel très proche du langage *OCaml* (la syntaxe de Coq est justement basée dessus).

La programmation fonctionnelle est basée sur le modèle de calcul du **Lambda calcul** qui exprime un programme comme une fonction mathématique (pouvant prendre une fonction en paramètre et renvoyer une fonction) plutôt qu'une suite d'instructions à exécuter.

Logique d'ordre supérieur

Il existe plusieurs systèmes permettant de représenter les expressions logiques qui nous permettent de former des raisonnements.

La **logique propositionnelle** ou *calcul propositionnel* nous permet d'exprimer des formules simples avec des variables booléennes reliées par des connecteurs logiques.

La **logique du premier ordre** ou *calcul des prédicats* nous permet de quantifier sur les éléments d'un univers. C'est à dire que exprimer qu'une propriété est vraie sur tous les éléments ou qu'il existe un élément tel qu'elle est vraie dans un univers précis. Cette logique intègre aussi des fonctions renvoyant un objet de l'univers et des prédicats qui permettent de mettre en relation plusieurs objets d'un univers. Exemple : $\phi = estHumain(frereDe(x))$. *frereDe* est une fonction et *humains* est un prédicat. ϕ prendra ensuite la valeur *Vrai* ou *Faux*.

La **logique d'ordre supérieur** ou *logique du second ordre* quant à elle, va nous permettre de quantifier sur les fonctions et prédicats et d'avoir des variables se référant à elles. Dans Coq cela nous permet d'exprimer qu'une propriété est vraie pour toute formule, pour toute fonction etc. Cela permet d'exprimer plus de choses que dans les deux systèmes précédents.

Logique intuitionniste

On distingue les formalismes logiques intuitionnistes et classiques.

La **logique classique** se fonde sur la notion de vérité alors que la **logique intuitionniste** se fonde sur la notion de preuve.

La conséquence est que certaines formules logiques sont vraies mais impossibles à prouver. On peut citer par exemple la formule de *tiers exclu* qui énonce que $P \vee \neg P$ c'est à dire que toute formule est soit vraie soit fausse. Coq construisant des preuves intuitionnistes il est impossible de prouver le tiers exclu et ses dérivés.

La logique intuitionniste est née par le biais de courants philosophiques concernant les mathématiques qui refusent les preuves classiques utilisant le tiers exclu ou ses dérivés.

Son intérêt est le suivant : imaginons que l'on veut prouver que quelque chose existe. Une preuve classique pourrait procéder par un **raisonnement par l'absurde** en supposant que la chose n'existe pas et en déduire une contradiction. Les mathématiciens intuitionnistes rejettent ce raisonnement car pour prouver l'existence il faut exhiber un témoins **(preuve par construction)**. Une preuve intuitionniste porte plus d'informations.

Restriction de terminaison

Les fonctions de Coq terminent toujours. Pour s'assurer de cela, les fonctions récursives de Coq posent des restrictions : dans une fonction récursive, toutes les variables dans l'appel récursif doivent être "structurellement plus petits" que le paramètre en entrée de la fonction.

Ainsi à chaque itérations on est certains qu'un paramètre décroît. Les types de Coq possédant une "base" (celle des entiers sera 0) on est certains (si on gère le cas de base) que la fonction va finir par s'arrêter (notion de relation bien fondée).

Note : si les fonctions de Coq autorisaient les récursions infinies, il serait possible de prouver qu'une contradiction est vraie ce qui remettrait en question la fiabilité de toutes les preuves de Coq.

Restriction de définition

Les fonctions de Coq sont définies pour toute valeur des types en paramètres.

Elles doivent gérer tous les cas possibles de manière exhaustive. Par exemple si on a une fonction qui prend en paramètre un entier, il faudra que pour tout entier, la fonction retourne quelque chose.

Correspondance de Curry-Howard

La *correspondance de Curry-Howard* ou *correspondance preuve-programme* est une liaison entre les propriétés calculatoires des systèmes fonctionnels (programmes) et des preuves mathématiques (logique).

Cette correspondance nous permet de voir les preuves mathématiques comme des programmes que l'on peut programmer mais aussi de voir les programmes fonctionnels comme des preuves de théorèmes mathématiques.

Cette liaison est le cœur du système de vérification de preuves de Coq. Prenons une fonction $f : T \mapsto T'$ prenant un paramètre de type T et renvoyant un objet de type T'. Une application de cette fonction à un paramètre x de type T donne un résultat $f(x)$ de type T'.

Les propriétés calculatoires du système de typage correspondent à la règle de déduction *Modus Ponens* de la logique qui dit que si on sait que A et $A \Rightarrow B$ sont vrais alors on peut déduire que B est vrai. On peut voir cette règle comme l'utilisation d'un lemme A pour prouver un théorème B.

Ainsi vérifier une preuve mathématique pourrait revenir à vérifier si une expression fonctionnelle est bien typée. La correspondance se traduit par les règles d'inférences suivantes :

$$\frac{f : T \to T' \quad x : T}{f(x) : T'} \text{ Application} \qquad \frac{T \Rightarrow T' \quad T}{T'} \text{ ModusPonens}$$

On remarque qu'une variable logique correspond aussi à un type. C'est justement comme ça que les expressions logiques sont représentées : comme des types qui contiennent leurs preuves.

Bibliographie

[1] B. C. Pierce, *Software Foundations*. University of Pennsylvania, Version 3.2, 2015.

[2] B. C. Pierce, *Types and programming languages*. Cambridge, Massachusetts, The MIT Press, 2002.

[3] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, B. C. Pierce, B. Eng *Beyond Good and Evil : Formalizing the Security Guarantees of Low-Level Compartmentalization*. 2016.

[4] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach., *Micro-policies : Formally verified, tag-based security monitors*, In 36th IEEE Symposium on Security and Privacy (Oakland S&P), 2015.

[5] M. Abadi., *Protection in programming-language translations.*, Research Report 154, SRC, 2015.

[6] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, Steve Zdancewic, *SoftBound : Highly Compatible and Complete Spatial Memory Safety for C* , University of Pennsylvania.

[7] X. Rival., *Operational Semantics - Semantics and applications to verification (Lecture slides)*, École Normale Supérieure, 2015. (Slides : `http://www.di.ens.fr/~rival/semverif-2015/sem-02-trace.pdf`)

[8] A. M. Pitts ., *Semantics of Programming Languages (Lecture notes)*, University of Cambridge, 2002. (Notes : `http://www.inf.ed.ac.uk/teaching/courses/lsi/sempl.pdf`)

[9] M. Laplaige., *Rapport de Stage deDUT Informatique*, IUT de Montreuil, Promotion 2014-2015.

[10] C. Hriţcu., *Micro-Policies : Formally Verified, Tag-Based Security Monitors (Talk Rennes)*, Inria Paris - Prosecco, 2015. (Slides : `http://prosecco.gforge.inria.fr/personal/hritcu/talks/Micro-Policies-Rennes.pdf`)

Sitographie

1. `https://fr.wikipedia.org/wiki/Plan_Calcul`
2. `https://fr.wikipedia.org/wiki/Institut_national_de_recherche_en_informatique_et_en_automatique`
3. `http://www.inria.fr/centre/paris/presentation/une-forte-reconnaissance`
4. `https://fr.wikipedia.org/wiki/IRILLS`
5. `http://prosecco.gforge.inria.fr/people.php`

6. http://www.inria.fr/institut/inria-en-bref/chiffres-cles
7. http://www.inria.fr/institut/strategie
8. http://www.inria.fr/institut/partenariats/partenariats-industriels
9. https://en.wikipedia.org/wiki/Coq
10. http://www.inria.fr/centre/paris/recherche

Glossaire

> Termes

MSR : Microsoft Research
CNRS : Centre National de Recherche Scientifique
Haut niveau : On dit qu'un langage est de haut niveau s'il s'abstrait des caractéristiques techniques d'une machine (Java, Python...). On ne peut pas déterminer objectivement si un langage est de haut niveau mais on peut dire qu'il est de plus haut niveau qu'un autre.
Bas niveau : Par opposition à la notion de haut niveau, on dit qu'un langage est de bas niveau s'il permet de manipuler des mécanismes proches de la machine comme la mémoire ou les registres.
Politique : Règle qui vise à maintenir un niveau de sécurité dans un système.
Micro-politique : Politique directement appliquée au niveau du micro-contrôleur.
Moniteur : Mécanisme qui vérifie si une violation des politiques se produit.
Compilation : Mécanisme de conversion d'un langage source vers un langage cible de plus bas niveau dans l'objectif de le faire exécuter par une machine.
Théorème : Expression logique prouvable par dérivation successive de règles logiques.
Lemme : C'est aussi un théorème mais sous un autre nom. On désigne par "lemme" un énoncé logique prouvable mais utilisé comme auxiliaire pour prouver un théorème qui est plus important.
Programmation impérative : Style de programmation où les opérations réalisées par un programme sont représentées par une séquence d'instructions exécutées successivement par une machine et modifiant l'état de celle-ci. Exemples : C, C++, Java, Python.
Programmation fonctionnelle : Style de programmation où l'objet élémentaire est la fonction. Les fonctions ne sont pas des suites d'instructions mais définies comme une relation entre le paramètre et le résultat comme pour une fonction mathématique. Les programmes fonctionnels ne possèdent pas de boucles par défaut, ils utilisent des appels récursifs de fonction. Exemple : OCaml, Haskell, F#.
Programmation orientée objet (POO) : Style de programmation qui découpe un programme en blocs appelés classes représentant des concepts abstraits ou concrets possédant des attributs et fonctions et fait interagir un programme en utilisant des instances de ces classes.
Grammaire BNF/Non-contextuelle : grammaire définissant une syntaxe de façon récursive. On parle de grammaire non-contextuelle ou algébrique en théorie des langages et de grammaire BNF dans un contexte de langages de programmation en particulier. **Opcode** : Codage associé à une instruction de bas niveau. Chaque instruction correspond à un

opcode qui est un identifiant sous forme d'entier binaire.
SCC : Secure Compartmentalizing Compilation. Nouvelle propriété de garantie de sécurité proposée par l'article «Beyond Good and Evil»
Divergence : si un programme diverge, selon signifie qu'il boucle à l'infini et ne s'arrête jamais.

> Notations

Programmes

Langage source

φ (*phi*) : programme partiel ou complet
κ (*kappa*) : composant
P : identifiant de procédure
b : identifiant de buffer
C : identifiant de composant
$\otimes$: opérateur binaire quelconque
$C.P$: appel de procédure d'identifiant P dans le composant c
ι (*iota*) : interface de composant
Ψ (*psi*) : interface de programme
s : état
cfg : configuration
δ (*delta*) : pile d'appel
K : continuation
Δ (*delta*) : contexte de valeurs
$\Delta \vdash c \rightarrow c'$: réduction de configuration
η (*eta*) : invariant de composant
Γ (*gamma*) : invariant de programme

Langage cible

mem : mémoire
reg : registres
pc : program counter, pointe sur l'instruction courante
r_{ra} : registre de retour d'appel
r_{com} : registre de communication inter-composants

Compilation

$\downarrow$: compilation
A : attaquant ou contexte (programme partiel)
$A[P]$: fusion de programmes partiels (application)

Sémantique de traces

a (*alpha*) : action interne ou externe

γ (*gamma*) : action externe (qu'on note aussi Ea)

$\gamma!$: action externe provenant du programme

$\gamma?$: action externe provenant du contexte

$\checkmark$: symbole de terminaison de programme

$P \in \bullet s$: le programme P a la forme s

$A \in \circ s$: le contexte A a la forme s

$Traces_{\circ s}(p)$: ensemble des traces du programme de bas niveau p de forme s

$Traces_{\bullet s}(a)$: ensemble des traces du contexte de bas niveau a de forme s

ζ (*zeta*) : fonction de canonisation (nettoyage des registres)

Secure Compartmentalizing Compilation

$\sim_H$: similitude des comportements (divergence/terminaison) pour deux programmes de haut niveau

$\sim_L$: similitude des comportements (divergence/terminaison) pour deux programmes de bas niveau

$\not\sim_H$: distinction des comportements (divergence/terminaison) pour deux programmes de haut niveau

$\not\sim_L$: distinction des comportements (divergence/terminaison) pour deux programmes de bas niveau

Logique

$\wedge$: ET logique (conjonction)

$\vee$: OU logique (disjonction)

$\neg$: NON logique (négation)

$\rightarrow$ ou $\Rightarrow$: Implication logique (Si...alors...)

$\forall$: Pour tout (quantificateur universel)

$\exists$: Il existe (quantificateur existentiel)

$\in$: appartenance à un ensemble

$\subseteq$: inclusion d'un ensemble dans un autre

Table des matières