Contrôle vérifié de flux d'information appliqué aux systèmes cyber-physiques

Jean-Joseph Marty

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Jean-Joseph MARTY

Contrôle vérifié de flux d'information appliqué aux systèmes cyberphysiques

Thèse présentée et soutenue à Rennes, le 17 novembre 2022
Unité de recherche : INRIA

Rapporteurs avant soutenance :

Marie-Laure POTET	Professeur des universités, IMAG
Catherine DUBOIS	Professeur des universités, ENSIIE

Composition du Jury :

Présidente :	Sandrine BLAZY	Professeur des universités, Université de Rennes
Rapporteurs :	Marie-Laure POTET	Professeur des universités, IMAG
	Catherine DUBOIS	Professeur des universités, ENSIIE
Examinateur de thèse :	Thomas JENSEN	Directeur de recherche, INRIA Rennes

Résumé

L'informatique est devenue omniprésente dans la vie de tous les jours au travers d'outils de plus en plus riches technologiquement. Généralement, ces objets – qualifiés d'*intelligent* et de plus en plus connectés à internet – aident à résoudre des tâches du quotidien dans des situations hétérogènes. Lorsque ces objets arrêtent de fonctionner suite à un concours de circonstances, cela peut avoir des conséquences dramatiques sur la sûreté des personnes qui en dépendent. Leur fonctionnement peut être aussi altéré par une volonté malveillante de nuire et qui lors d'une attaque créera la situation propice à un dysfonctionnement ciblé.

Le risque d'un échec de tels objets questionne sur leur criticité en termes de sûreté ainsi qu'en termes de sécurité. En plus des erreurs logicielles, il existe de nombreux exemples d'erreurs liées à la lecture de capteurs ou à la modification d'actionneurs. À la frontière avec le monde physique, cette partie du logiciel embarqué – appelée le monde cyberphysique – décrit comment un ordinateur perçoit et modifie l'environnement dans lequel il opère.

Dans ce travail, nous explorons l'utilisation des méthodes formelles dans un environnement embarqué et cyberphysique. Dans un premier temps, nous utilisons F⋆ pour modéliser la sûreté des programmes sur une Arduino. Puis nous implémentons Library Input Output (LIO) en F⋆ dans un contexte système ou embarqué. Nous mettons ici en avant une approche qui permet de choisir entre une vérification statique et une vérification dynamique afin de réduire la charge de preuve ou le coût à l'exécution. Nous proposons également une automatisation de la preuve de non-interférence en utilisant la métaprogrammation. Cette preuve porte sur l'interaction entre le programme et la bibliothèque plutôt que sur la bibliothèque seulement.

Remerciements

Cette thèse a été pour moi un merveilleux voyage dans le monde scientifique, mais aussi dans celui des relations humaines. En effet, j'ai dû faire face à de nombreux obstacles que je n'aurais probablement pas pu surmonter sans l'appui et le soutien de nombreuses personnes.

Je tiens tout d'abord à remercier Lucas Franceshino qui m'a énormément aidé à façonner certains aspects essentiels de mon travail et avec qui travailler a été un plaisir, ainsi que Niki Vazou dont les conseils sur la preuve de non-interférence et la rédaction de papiers scientifiques ont été déterminants.

De même, la soutenance n'aurait jamais eu lieu sans les retours de Thomas Jensen, lequel a permis que le jury puisse lire la thèse et y apporter une critique.

Mes remerciements vont également aux membres de ce dernier qui ont montré leur intérêt et leur lecture approfondie du manuscrit par leurs remarques constructives.

Simon Lunel, Alexandre Honorat, Lucas Franceshino et Stéphane Kastembaum ont été membres de l'équipe de recherche dans laquelle a été commencé ce travail de thèse et m'ont donné la force d'aller jusqu'à la fin d'un tel travail grâce à la qualité de leurs échanges scientifiques et les moments chaleureux partagés ensemble.

Bien qu'en distanciel, car vivant dans une autre ville, Rémy Pasquion et Quentin Ladugie ont été présent pendant les épreuves et les confinements, ce qui fut très important pour moi.

Je suis reconnaissant à la société Amossys d'avoir accepté que je termine ce travail de fin de thèse en parallèle de mon travail professionnel me permettant d'aboutir à un manuscrit finalisé et une soutenance.

Mes pensées vont particulièrement à toutes les personnes que je ne cite pas, mais qui ont eu un rôle important pour moi et qui le savent.

Plus que tout, je remercie mes parents, ma famille et mes amis de m'avoir enseigné des valeurs qui m'ont aidé à être résilient face aux épreuves de la vie.

Table des matières

Table des figures

Liste des tableaux

Acronymes

HACL⋆ High-Assurance Cryptographic Library in F⋆. 37

ASIC Application-specific integrated circuit. 6

AST Abstract Syntax Tree. 186, 187

BIOS Basic Input/Output System. 108

CFI Contrôle de Flux d'Information. 3, 12, 14–16, 18–21, 23–26, 123, 125, 126, 128–135, 137, 138, 142, 143, 145–147, 151, 154, 155, 157, 160–162, 164–166, 168, 172, 174, 175, 183, 184, 188–196

CPS Cyber-Physical System. 8, 192–194

CPU Central Processing Unit. 6, 7

DAC Discretionary Access Control. 11

DSL Domain Specific Language. 9

DSP Digital signal processor. 6

FPGA Field-programmable gate array. 6

GCC Gnu C Compiler. 64, 65

GHC Glasgow Haskell Compiler. 162

GPS Global Positioning System. 8

GPU Graphical Processing Unit. 5

HLIO Hybrid LIO. 17, 195

IPC Inter-Process Communication. 126, 139

LIO Library Input Output. i, 12, 16–18, 20–24, 26, 123, 125, 130–134, 136, 138–140, 142, 143, 145–149, 155–157, 159–162, 166–168, 172, 173, 176, 177, 183, 185, 188, 189, 191, 192, 195

LTL Linear Temporal Logic. 9

MAC Mandatory access control. 11, 16, 18

MCAS Maneuvering Characteristics Augmentation System. 86, 87

MMU Memory Management Unit. 24, 26, 108–115, 120, 123, 160, 192–194

MPU Microcontroler Processing Unit. 6–9

PWM Pulse Width Modulation. 84, 88, 93, 94

RBAC Role-based access control. 11

RISC Reduced Instruction Set Computer. 109

RTOS Real Time Operating System. 8

SMT Satisfiability Modulo Theory. 37, 42–44, 53, 55–57, 62, 63, 65, 128, 196

TCB Trusted Computing Base. 130, 132, 143, 144

TCSEC Trusted Computer System Evaluation Criteria. 11

TLS Transport Layer Security. 37

Préambule

Depuis l'invention de l'ordinateur, nous l'intégrons de plus en plus dans les technologies de domaines très hétérogènes. L'apport d'un tel appareil nous permet d'augmenter la précision et les fonctionnalités de nos outils. Cependant, l'ordinateur ajoute une complexité importante : celle du logiciel. En effet, écrire un logiciel peut s'avérer très compliqué, tant pour ce qui concerne la conception d'un algorithme, que vis-à-vis de ses interactions, et pour assurer son bon fonctionnement. Dans ce document, nous nous focalisons sur les assurances que nous pouvons avoir sur le bon fonctionnement d'algorithmes ainsi que leurs interactions avec le monde physique. Pour illustrer l'impact de l'informatique, remontons le temps avec un article (Witt 2020) qui retrace l'arrivée de l'homme sur la lune. Nous avons retranscrit ici une partie de l'histoire afin de montrer comment le logiciel peut avoir un impact essentiel dans nos vies.

Le 20 juillet 1969, deux hommes – Neil Armstrong et Buzz Aldrin – sont sur le point de se poser sur la lune. Après s'être détaché du module de commande, le vaisseau *Eagle* entame sa descente. Au bout de quelques minutes, Aldrin prend des mesures au radar pour comparer la position calculée. Une alarme retentit et l'ordinateur de bord retourne une erreur : 1202. Bien que les deux astronautes aient été entraînés à de nombreux scénarios, ils ne savaient pas ce que cela signifiait et encore moins que l'ordinateur de bord venait de planter.

Le centre de contrôle regarda la liste des codes d'erreur, il put rassurer l'équipage d'Apollo 11 : le message signifiait que l'ordinateur avait réussi à sauvegarder les données de navigation de l'atterrisseur avant de planter. La mission fut donc maintenue. Cependant, quelques minutes plus tard, à moins de 600 mètres d'altitude, l'alarme s'est déclenchée à nouveau, mais cette fois-ci l'affichage de l'atterrisseur s'est éteint pendant 10 secondes et après avoir redémarré, l'ordinateur indiqua encore l'erreur 1202. La situation ne s'est pas améliorée. Le système plantera 5 fois en 4 minutes. Mais abandonner en étant si proche de la surface rendait la procédure d'annulation beaucoup plus risquée.

Ces erreurs compliquèrent l'alunissage d'autant plus que le vaisseau ne se posa pas sur la zone initialement prévue (Armstrong s'était entraîné sur des photos de la zone d'alunissage pour savoir quels endroits étaient propices). Le vaisseau toucha le sol lunaire 40 secondes après un redémarrage de l'ordinateur. Plus tard, Armstrong confiera qu'il envisagea, pendant cet alunissage, la possibilité qu'il ne puisse jamais rentrer sur Terre. Que se passa-t-il lors de ce premier voyage guidé par ordinateur ?

Pour comprendre ce qui s'est passé, nous devons comprendre comment le logiciel de l'ordinateur de bord était conçu. Le système d'exploitation fut conçu par Hal Laning. La principale contrainte était que les entrées/sorties soient gérées en temps réel – c'est-à-dire sans aucun délai. Cette approche étant novatrice, il décida que chaque programme aurait une priorité qui déter-

mine si un programme peut ou non être interrompu. Cela permit d'avoir une sorte de processeur multi-tâches. Lors de l'implémentation par Charles Muntz, qui était un collègue de Laning, une limite est survenue : comment gérer la surconsommation de ressources ?

En effet, une saturation des ressources de l'ordinateur provoquerait un ralentissement de l'ordinateur, ce qui ferait perdre l'aspect temps réel du système d'exploitation. La solution de Muntz fut de sauvegarder les données des programmes indispensables au fonctionnement du vaisseau, redémarrer le système puis abandonner les programmes non indispensables. Il s'agit de cette erreur 1202. Lors de l'alunissage, le guidage nécessitait 87 % de ressources et les demandes d'Aldrin consommaient 3 %. Les ressources restantes étaient utilisées pour le radar de rendez-vous qui était actif afin de pouvoir guider le vaisseau vers le module de commande dans le cas d'une annulation de mission. Un bug fut trouvé dans ce programme qui de temps en temps envoyait des requêtes inutiles. Grâce à la procédure de redémarrage ces requêtes furent ignorées.

Ce bug était impossible à corriger dans la mission Apollo 11, car les programmes étaient du "cousu main" et donc non modifiables. Un des développeurs du logiciel de descente, Don Eyle, confiera l'inquiétude qu'il ressentit en suivant les communications entre le contrôle de mission et les astronautes. Connaissant le logiciel, il savait qu'il ne s'agissait pas d'une erreur mais d'une cascade de dysfonctionnements. Pour lui, l'annulation était la seule issue puisque plus le vaisseau se rapprocherait de la surface, plus l'ordinateur redémarrerait et la situation empirerait. Il écrira dans ses mémoires : "*Une chose terrible est active dans notre ordinateur, et nous ne savons pas ce que c'est, ni ce qu'elle va faire ensuite*". Dans l'entretien au journal Wired (Witt 2020), il conclura "*la mission a atterri, donc ils ont dû bien faire les choses*".

Aujourd'hui nous savons qu'Armstrong a pris le contrôle partiel du vaisseau pendant la descente, ce qui a réduit la charge de traitement de l'ordinateur. Toutefois, sommes-nous en mesure de pouvoir contourner l'outil informatique de la même manière dans les outils actuels ? Quoi qu'il en soit, il existe de nombreuses solutions pour connaître le comportement des programmes avant leur exécution.

L'évènement historique Apollo 11 pose l'objectif de notre thématique : la vérification de programmes. Cela consiste à s'assurer de la justesse de fonctionnement des programmes. Ce travail de vérification se décompose en 3 étapes : nous partons d'un algorithme qui est une abstraction de la réalité informatique ; nous modélisons ses propriétés pour raisonner dessus ; nous le concrétisons enfin en un programme concret compréhensible par la machine informatique.

Introduction

Nous présentons dans un premier temps quelques définitions, avant d'aborder un rapide historique des deux axes de ce travail : l'informatique embarquée et le Contrôle de Flux d'Information (CFI). Ensuite, nous discuterons des questions de recherche avant de lister les contributions. Enfin, nous conclurons par un plan de lecture qui a pour objectif de montrer les dépendances de chaque chapitre.

Définitions

Dans cette section, nous présentons succinctement des termes qui sont souvent utilisés dans ce manuscrit. Ces définitions générales n'ont pas vocation à définir formellement les notions sous-jacentes. Une définition formelle est plutôt fournie par des références aux travaux de l'état de l'art ou par des illustrations dans les chapitres concernés.

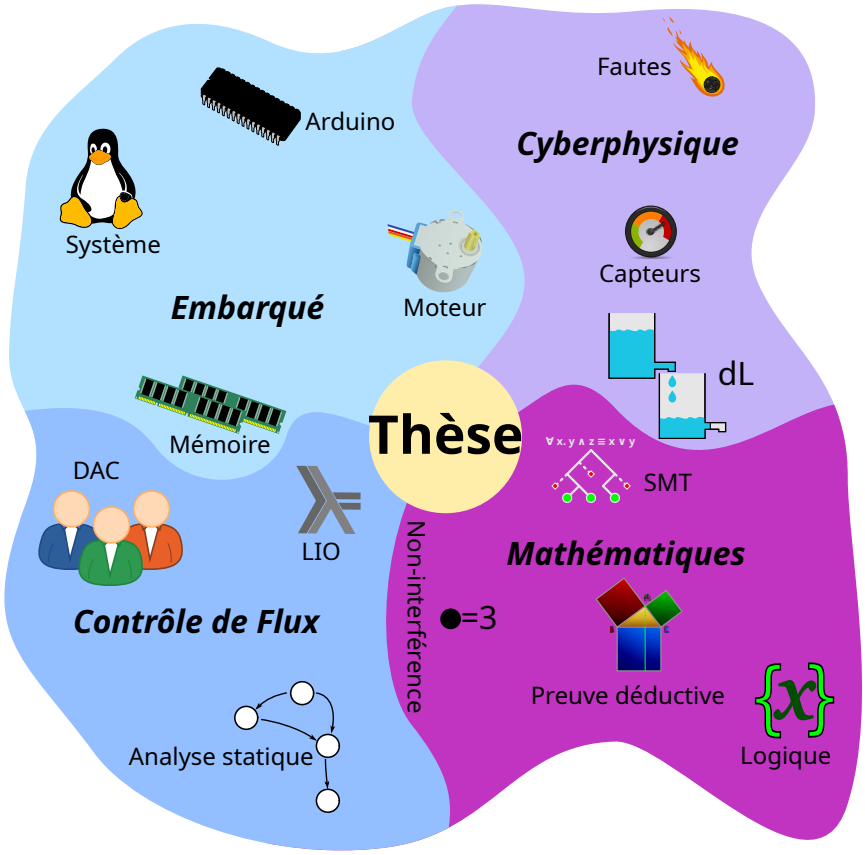

FIGURE 1 – Domaines abordés dans notre travail

Nous représentons sur la figure 1 les différents domaines que nous approchons dans nos

travaux. Notre sujet est situé au carrefour de quatre directions : l'informatique système, les systèmes cyberphysiques, le contrôle de flux d'information et la vérification formelle appliquée.

Vérification de programmes Présenté en page 1, l'exemple de la course à la lune est représentatif du danger qu'un programme peut engendrer. La vérification dans un sens large couvre l'intégralité des méthodes pour assurer le bon fonctionnement d'un programme. La relecture – qui est une méthode de vérification d'un programme – est souvent insuffisante, car elle dépend de l'expérience du développeur qui relit et de la complexité du programme. Pour avoir des garanties solides, il existe des solutions, comme l'analyse statique, qui permettent d'évaluer un programme vis-à-vis de certaines propriétés. Nous distinguons deux stratégies : la simulation [1] et la preuve.

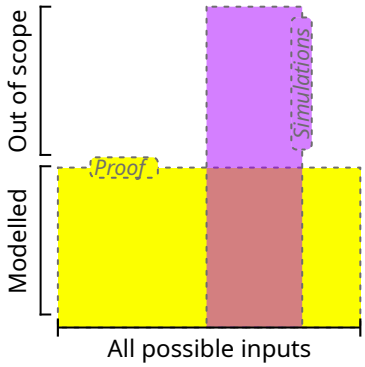

FIGURE 2 – Couverture de la simulation et couverture de la preuve

La simulation a pour avantage de tester un programme dans un environnement. Cela permet de tester l'interaction du programme avec ses entrées/sorties et avec les autres composants, mais cela présente un coût assez conséquent. En effet, simuler un programme nécessite de tester l'ensemble des valeurs possibles. C'est pourquoi les tests ne sont pas exhaustifs et cette limitation empêche de vérifier le programme pour toutes les valeurs possibles.

La preuve a pour avantage d'assurer des propriétés pour l'ensemble des valeurs possibles d'un programme. Cela demande une modélisation, ce qui permet de comprendre le comportement du programme. Cependant, cette modélisation concerne généralement le seul programme et pas son interaction avec ses entrées/sorties et les autres composants. En effet, plus une modélisation est précise, plus elle sera verbeuse et complexe, ce qui a un coût important.

La figure 2 représente graphiquement les avantages et inconvénients de chacune des stratégies. La simulation aura une couverture plus profonde – voire totale, si elle est utilisée en condition réelle – et la preuve aura une couverture plus large. Plus la "couverture" est importante, plus l'effort sera coûteux et précis. Ainsi, le coût de chaque stratégie peut être modulé afin de varier leur couverture sur l'axe limitant – vertical pour la simulation et horizontal pour

1. Dans ce contexte, la simulation désigne une exécution en situation réelle. Plus loin dans ce manuscrit, nous utilisons le terme "preuve par simulation" qui est différent. En effet, il s'agit d'une méthodologie de preuve qui n'exécute pas réellement les programmes.

la preuve. Cependant, ces deux stratégies sont tributaires de l'environnement et de la précision de sa modélisation.

Sécurité et sûreté Dans un programme, l'échec provient d'une erreur qui peut avoir plusieurs origines. Par exemple, elle peut venir d'une situation non gérée et nous dirons que le programme n'est pas sûr. Dans ce cas, le programme n'a pas été vérifié pour cette situation. C'est pourquoi la spécification doit être suffisamment précise pour prendre en compte la réalité de l'exécution d'un programme. Cependant, la malchance peut être au rendez-vous, laissant apparaître le manque de couverture de la vérification. La sûreté peut être résumée de la manière suivante : "*un programme sûr respecte **au moins** la spécification*".

Toutefois, une erreur peut être causée par un acte de malveillance et nous dirons que le programme n'est pas sécurisé. Dans ce cas, la spécification aura été étudiée par l'attaquant pour discerner les points faibles. C'est pourquoi des garde-fous supplémentaires sont nécessaires pour assurer que le programme n'a pas de capacités additionnelles. En effet, selon son pouvoir, un attaquant peut altérer l'environnement d'exécution afin de créer une situation favorable. En cas d'attaque, la chance ne sera probablement pas au rendez-vous puisque l'attaquant utilisera la spécification contre le programme lui-même : il peut à la fois utiliser un manque de sûreté et utiliser un comportement non défini. La sécurité peut être résumée de la manière suivante : "*un programme sécurisé respecte **au plus** la spécification*".

La sécurité est un terme beaucoup plus large que la sûreté puisqu'il englobe les situations imprévues et les situations artificielles créées par un attaquant. De manière évidente, plus un attaquant est fort dans un modèle d'attaque, plus il sera difficile de le contrer. Ainsi, lors d'une rédaction de spécification, il faut prendre en compte le pouvoir que peut avoir l'attaquant.

Présentation historique de l'informatique embarquée

Si nous demandons à des personnes au hasard ce que représente pour elles l'utilisation de l'informatique dans la vie de tous les jours, il est probable qu'elles parlent de navigation internet, de support multimédia ou même d'outil indispensable pour leur travail. Beaucoup n'incluront pas leur micro-onde, lave-vaisselle ou box internet dans les appareils informatiques qui pourtant leur apportent un grand confort.

Ces machines qui nous entourent sont dédiées à une tâche plus ou moins complexe : faire du café, gérer les différents paramètres du moteur d'une voiture… Pendant longtemps un ordinateur était conçu pour une tâche unique et son utilisation se limitait à cette tâche. De nos jours nous pouvons jouir d'un certain luxe en utilisant des ordinateurs extrêmement polyvalents dans leur fonctionnement. Les Graphical Processing Unit (GPU) – conçus pour gérer l'affichage – sont surtout spécialisés dans le calcul de données volumineuses et servent par exemple à entraîner des réseaux de neurones artificiels en même temps que calculer une représentation 2D de modèles

3D. Toutefois, ce type de composant est surdimensionné par rapport à un composant dédié à résoudre un seul problème.

À l'origine, les premières machines programmables ne l'étaient que pour des situations très précises comme les *bomba kryptologiczna* et les ordinateurs *Colossus* pendant la Seconde Guerre mondiale ((S. Singh 2000), (Jiménez, Palomera et Couvertier 2013)). Hélas ce genre de calculateur comportait deux problèmes majeurs : leur coût et leur manque de flexibilité. En effet, produire une architecture pour résoudre un problème spécifique représente un travail immense puisque des ingénieurs doivent concevoir la machine puis des industriels doivent la construire. Cette approche coûte cher, peu de modèles sont construits et elle est peu flexible puisque résoudre un autre problème nécessitera une architecture différente. Il est important de rappeler que dans certains domaines militaires ou scientifiques c'est toujours le cas. Il est parfois nécessaire de construire des machines sur mesure pour répondre à des impératifs sécuritaires ou de performances (Jiménez, Palomera et Couvertier 2013). Dans ce dernier cas, ces machines auront des composants dédiés de type Application-specific integrated circuit (ASIC), Digital signal processor (DSP) ou Field-programmable gate array (FPGA) (Jiménez, Palomera et Couvertier 2013).

En conséquence, l'informatique s'est diversifiée en deux branches majeures qui répondent aux deux limites des premières machines : l'une privilégiant la flexibilité, qui a donné des composants hautes-performances et polyvalents appelés Central Processing Unit (CPU), et l'autre privilégiant le coût, qui a donné des composants peu performants et peu flexibles mais bon marché, Microcontroler Processing Unit (MPU). La première approche correspond aux ordinateurs et smartphones que nous utilisons tous les jours. La seconde correspond aux microcontrôleurs qui sont nécessaires pour un grand nombre d'objets que nous utilisons au quotidien.

Grâce à l'informatisation des différents outils et moyens de transport, nous pouvons utiliser des moyens de plus en plus sophistiqués avec de moins en moins de nécessité à connaître le fonctionnement interne de la machine. Cependant, il serait dangereux de considérer que nous n'avons plus à comprendre ce fonctionnement. En effet, la question qui se pose alors est à quel point les décisions prises par un ordinateur de bord sont-elles fiables ? Dans cette section, nous répondrons à cette question en mettant l'accent sur l'interconnexion entre le monde physique qui, par notre limite de compréhension, est imprévisible, et le monde logique de l'informatique qui, par notre choix de construction, est prévisible.

Informatique embarquée

Dans l'exemple d'Apollo 11 en page 1, le système d'exploitation mélange programmes et ressources. En général un système d'exploitation permet le bon partage des ressources entre les programmes : il configure et entretient l'environnement d'exécution des programmes. La programmation à ce stade doit prendre en compte la spécificité de l'environnement et donc du système d'exploitation. Lorsque nous parlons de programmation système – également appe-

lée *bas niveau* par opposition à la programmation *haut niveau* où les abstractions rendent un programme indépendant des détails techniques – il s'agit soit de système d'exploitation gérant des ressources matérielles, soit de programme interagissant avec le système d'exploitation et sa gestion de ressources.

Le livre (Jiménez, Palomera et Couvertier 2013) indique que la première utilisation du terme *système embarqué* viendrait du programme d'exploration spatiale Apollo dans les années 60. Cet ordinateur était très loin des microcontrôleurs puisque son circuit avait été conçu pour des utilisations très spécifiques. En effet, pour être facilement utilisable, le composant responsable de l'exécution du logiciel doit avoir un minimum de flexibilité. C'est pourquoi l'arrivée des microprocesseurs dans les années 70 va changer la donne.

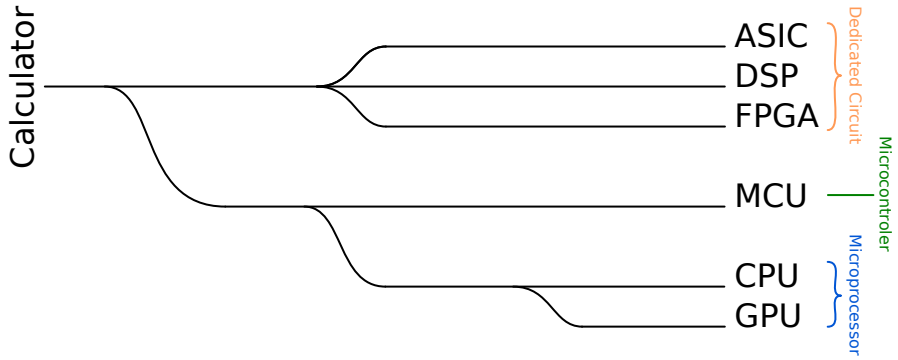

Figure 3 – Taxonomie des machines informatiques

Ces nouveaux processeurs vont permettre d'avoir des composants génériques. Leur production en grand nombre va aboutir à une réduction de leur coût de fabrication et ils engendreront les CPU que nous utilisons de nos jours dans les ordinateurs, cela est illustré sur la figure 3. Cependant, certains d'entre eux ne bénéficieront pas des avancées que nous avons dans nos ordinateurs, car ils seront destinés à des tâches simples où l'informatique doit avoir un impact minimum tout en étant responsable du bon fonctionnement. Ces microprocesseurs sont adaptés pour interagir avec leur environnement : ce sont les MPU (Camposano et Wilberg 1996). En effet, un MPU est un ordinateur complet. Il possède des entrées/sorties, une mémoire et un système de stockage permanent.

Nous pouvons définir l'informatique embarquée comme une approche de l'informatique dans laquelle l'ordinateur décisionnaire fait partie intégrante de l'objet (Marwedel 2006). De plus, l'informatique embarquée, à cause des contraintes de son embarquement, se distingue de l'informatique classique : il est nécessaire de privilégier l'intégration dans l'objet plutôt que la polyvalence. C'est principalement la raison qui fait que les MPU sont en retard en termes de fonctionnalités, cependant ils sont meilleurs en termes de gestion de l'énergie par exemple.

Systèmes *cyberphysiques*

Le système embarqué a des entrées/sorties qui sont propres à une conception. En effet, le but d'un système embarqué étant de répondre à une tâche spécifique, sa conception est propre à son utilisation. Ainsi, les capteurs correspondent à un besoin et sont utilisés pour répondre à une tâche. Un système embarqué peut percevoir l'environnement via des capteurs – un Global Positioning System (GPS) par exemple – et il peut le modifier via des actionneurs – un moteur par exemple. Cependant, le fait d'interagir avec l'environnement rend le programme *cyberphysique*. En effet, un robot qui se déplace change son référentiel et change sa perception de l'environnement. Contrairement à l'ordinateur de bord qui donne des informations ou applique des opérations de maintenance, le système *cyberphysique* a un impact qu'il doit prendre en compte. Les systèmes *cyberphysiques* sont souvent assimilés à des systèmes embarqués – c'est une définition plus générique. Dans nos travaux, nous soulignons cette nuance.

Ces derniers sont des MPU qui prennent en compte et interagissent avec des processus physiques (Marwedel 2006). Par exemple, l'évolution du taux de sucres dépend de nombreux facteurs dans un corps humain et une pompe à insuline pourra le calculer de manière précise en fonction de ce qu'indique le patient. Cela permettra à la machine de donner plus efficacement une dose d'insuline en fonction du moment de la journée mais aussi de l'activité du patient. N'oublions pas qu'un tel dispositif doit nécessairement afficher des informations correctes et être capable de récupérer les bonnes informations provenant des capteurs qui peuvent, selon l'environnement, perdre en efficacité (J. Wang et al. 2007) (Zhang et al. 2011).

Un autre exemple est le cœur qui peut être équipé d'un *pacemaker*. Ce dispositif est très critique et fonctionne à l'intérieur du corps humain (N. K. Singh, Wellings et Cavalcanti 2012). Son objectif est d'envoyer une impulsion électrique lorsqu'un certain laps de temps s'est écoulé. Ce dernier modifie son environnement, tient compte du fonctionnement biologique de l'être humain et réagit en conséquence.

Ainsi la modélisation de l'environnement est essentielle pour intégrer des aléas naturels venant du monde physique dans la conception du programme. Les Cyber-Physical System (CPS) sont à la frontière entre l'informatique embarquée et le monde physique puisqu'ils permettent de faire le lien entre l'interprétation de l'environnement physique par le logiciel et la manière dont lui-même le modifie.

Contraintes

Le monde de l'embarqué propose des problèmes très hétérogènes qui poussent les développeurs à repenser la manière d'écrire un programme. Dans un ordinateur, le système d'exploitation gère les ressources qui ont des priorités différentes. Une différence entre le système d'un ordinateur normal et celui d'un système temps réel embarqué – Real Time Operating System (RTOS) – peut être, par exemple, le temps de réponse. Ce dernier est non-déterministe dans un système normal contrairement au système embarqué où il est déterministe (*What is an RTOS ?*

2020).

En plus des contraintes liées au matériel, les programmes embarqués doivent tenir compte de la spécificité des microcontrôleurs sur lesquels s'exécute le programme. Ainsi de nombreux projets de Domain Specific Language (DSL) extraient leurs programmes en langage système – généralement C ou Ada – puisque les MPU sont extrêmement limités. En effet, certains de ces microcontrôleurs sont conçus pour fonctionner avec peu de ressources énergétiques et donc ont peu de capacités matérielles. Comme dit précédemment, les microcontrôleurs favorisent l'adaptation à la situation et préféreront la réactivité plutôt que la puissance, par exemple. De fait, les microcontrôleurs sont conçus pour être utilisables dans des scénarios spécifiques comme le montre la gamme des MPU de la société Atmel (*Chip Hall of Fame : Atmel ATmega8* 2017).

Dans cette thèse nous utilisons l'Arduino : un microcontrôleur Atmel328p 8 bits. Ce microcontrôleur possède une mémoire vive de 2 Ko et 32 Ko de stockage (*Atmel-7810-Automotive-Microcontrollers-ATmega328P Datasheet* 2020) (*List of Arduino boards and compatible systems* 2020). Nous avons choisi ce microcontrôleur parce qu'il est très répandu dans de nombreux projets grâce à son coût faible bien qu'il possède peu de mécanismes de sécurité.

Vérification formelle dans l'embarqué

L'informatique s'embarquant dans des objets de plus en plus complexes, il est devenu nécessaire d'avoir des garanties. Nous avons évoqué l'expressivité des langages pour rendre plus compréhensibles les programmes. Cependant, l'expressivité – comme avec un typage fort – peut être utilisée pour mettre en avant certaines propriétés. Cela aide à savoir si un système contient une erreur ou pas. Grâce à cela, nous nous éloignons du constat alarmant de la vérification dans l'embarqué en 1999 : "*Espérez et priez !*" (Colnaric 1999).

En 1999, l'article (Colnaric 1999) constate que les programmes embarqués dans le domaine du temps réel sont généralement peu ou pas du tout vérifiés sauf pour les applications critiques. Ce manque de vérification continuera comme le montrent de nombreux exemples que nous citerons – par exemple (Cummings 2016) et (Johnston et Harris 2019). Cependant, il soulève également une volonté d'apporter des solutions suite à des accidents comme celui de la fusée Ariane 5 (Robinson 2011).

Toutefois, il y avait déjà à cette époque de la vérification formelle dans l'informatique embarquée. Nous pouvons citer le projet *Météor* (Behm et al. 1999) qui a permis à la ligne 14 du métro parisien d'être vérifiée avec le langage B. Nous pouvons citer également l'outil *Penelope* qui permet d'augmenter le langage Ada avec des prédicats pour pouvoir vérifier la cohérence entre un programme et son comportement (Guaspari, Marceau et Polak 1990). Finalement, ce que l'article (Colnaric 1999) traduit, c'est un fossé entre l'industrie et la recherche : la vérification formelle est réservée à un univers restreint d'experts sur des solutions très spécifiques. Cette restriction vient de la complexité ajoutée au développement comme le conclut un article trois ans plus tôt (Espiau, Kapellos et Jourdan 1996). Ce dernier étudie la vérification formelle

sur des applications robotiques avec la Linear Temporal Logic (LTL).

Plus tard en 2005, l'utilisation des méthodes formelles est plus répandue. L'article (Loghi et al. 2005) résume l'état de l'art pour plusieurs approches de vérification dans les systèmes embarqués : la simulation d'exécution et la preuve de programmes. À ce moment la simulation d'exécution et la preuve de programmes – comme la vérification de modèle – semblent être favorisées notamment grâce à une meilleure appréciation par les ingénieurs industriels.

Enfin en 2019, un article (Souri et Norouzi 2019) donne une idée de l'évolution de la vérification dans le monde des objets connectés. Nous y découvrons que l'algèbre de processus a finalement progressé au point de dépasser la preuve formelle. Cependant, ce résultat est biaisé puisque l'article étudie les méthodes formelles automatisées qui représentent à ce moment un domaine nouveau : il est possible d'utiliser le calcul de la "précondition la plus faible" pour rechercher avec un solveur SMT les incohérences entre la spécification et l'implémentation (Kosmatov et Signoles 2016).

Dans cette thèse, nous explorons et étendons la vérification formelle dans le cadre de l'embarqué, tout en utilisant l'aspect automatique du calcul de la précondition la plus, en utilisant F⋆.

Présentation historique du CFI

Dans la vie courante, nous renseignons nos informations personnelles sur de nombreux sites web. Cependant, l'histoire nous montre que des fuites d'information peuvent rendre ces informations – à l'origine privées – accessibles à tous et donc publiques. Nous pouvons prendre l'exemple de "*At the pool*" – une application Facebook – qui collecte les données privées Facebook de ses utilisateurs afin de donner des suggestions de rencontres en fonction des centres d'intérêts mais aussi des activités potentielles ou en cours de l'utilisateur. Ces données concernent les amis, les *likes* ainsi que des mots de passe. Ces données ont pu être récupérées en clair grâce à une sauvegarde de ce service dans le cloud d'Amazon qui était public (*Losing Face : Two More Cases of Third-Party Facebook App Data Exposure* 2020). Même si cet exemple montre une erreur humaine, comment des attaquants manipulent-ils les systèmes informatiques pour transformer et divulguer ces informations ?

Parfois, le but d'une attaque informatique est de voler des informations pour les revendre. L'article (Hauer 2015) décrit un motif typique en trois étapes lors de chaque attaque : collecte de données, conversion afin de les cacher puis transfert discret. La première étape consiste à atteindre le système où les données sont stockées : c'est la collecte. Pour réussir cette étape, un attaquant doit réussir à augmenter ses privilèges pour accéder à des zones protégées du système informatique. La seconde étape a pour but de transformer l'information de manière à ce qu'elle soit considérée comme une information publique plutôt que sensible : la donnée est convertie pour ne pas être traçable. Ce changement de sensibilité correspond à la fuite à proprement parler. La troisième étape correspond à l'évasion de l'information. Le but est alors de faire transférer

les données en utilisant des moyens légitimes.

Une solution est d'empêcher l'accès selon les utilisateurs. Appelée le contrôle d'accès, cette approche est utilisée dans l'accès aux ressources d'un système d'exploitation puisqu'elle a coût négligeable – juste le test d'une autorisation pour un accès et un utilisateur donné.

Contrôle d'accès

Comme dit précédemment, cette approche est utilisée dans le contrôle des ressources, ce qui se prête bien aux systèmes d'exploitation. Le système définit alors des propriétaires qui peuvent en fonction de leurs droits accéder à des fonctions spécifiques. L'attaquant ne peut pas en théorie contourner ce contrôle – en pratique ce n'est absolument pas le cas ((Xiao et al. 2016), (Kocher et al. 2019), (Lipp et al. 2018)) mais nous sortons du cadre de ce travail. Nous pouvons citer le *livre orange* – appelé Trusted Computer System Evaluation Criteria (TCSEC) – qui liste les différentes approches techniques pour protéger les ressources d'un système d'exploitation (*Trusted Computer System Evaluation Criteria* 2020). Écrit dans les années 80, cet ouvrage témoigne de l'intérêt croissant à protéger l'accès à l'information (*Trusted Computer System Evaluation Criteria* 2020).

En 1976, le papier (Harrison, Ruzzo et Ullman 1976) propose de formaliser les contrôles d'accès avec des matrices d'accès. Sous cette forme, elles permettent pour chaque fonctionnalité de savoir qui peut être l'expéditeur et qui peut être le destinataire d'un flux. Cela a été le point de départ de nombreuses méthodes de contrôle d'accès comme le Discretionary Access Control (DAC) qui permet au propriétaire de choisir les droits d'accès aux utilisateurs (*Trusted Computer System Evaluation Criteria* 2020). Ces méthodes changent la manière dont le flux est exprimé : il peut être exprimé en fonction du rôle de l'utilisateur dans une organisation – comme Role-based access control (RBAC) (Sandhu et Munawer 1998) – ou simplement en fonction de son niveau de sensibilité – par exemple Mandatory access control (MAC) (Russo 2015). Ces variantes de contrôle d'accès sont utilisées dans le système d'exploitation Linux (*Mandatory access control* 2020).

Pour rappel, MAC permet à un utilisateur de lire uniquement dans des ressources de privilège inférieur ou égal à son niveau de sensibilité, et d'écrire uniquement dans des ressources de privilège supérieur ou égal à son niveau de sensibilité.

	Alice	Bob	Charles
Alice	Oui	Oui	-
Bob	-	Oui	Oui
Charles	-	-	-

Table 1 : Matrice d'accès.

Le contrôle d'accès a l'avantage d'être simple à implémenter. Une fois un utilisateur identi-

fié, il est possible de conditionner l'accès au reste du système. Cependant, cette approche n'est pas transitive : si un utilisateur accède à une donnée, il devient propriétaire de l'information et peut la redistribuer comme il le souhaite. Cela s'explique facilement puisque seule la validité de l'accès est contrôlée et non pas la manipulation. C'est à ce niveau que les flux d'information prennent sens. En fait, manipuler une information revient à lier plusieurs données entre elles ((Cohen 1977), (Goguen et Meseguer 1982)). De même, le transfert de l'information entre deux acteurs est un flux.

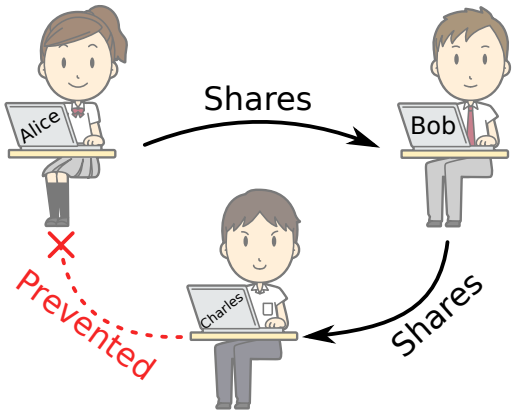

Figure 4 – Fuite de données et transitivité

Imaginons le scénario où nous avons trois acteurs : *Alice*, *Bob* et *Charles*. Ce dernier est malveillant et souhaite lire des données en provenance d'*Alice*, ce que cette dernière ne veut pas. Grâce au contrôle d'accès, il ne peut pas accéder aux données. Cependant, *Bob* travaille avec *Alice* et ils partagent des données entre eux. En convainquant *Bob*, *Charles* peut lui demander de transmettre les informations d'*Alice* auquelles il a accès : il s'agit d'une transmission transitive. Nous illustrons de tels flux avec la figure 4 et nous donnons la matrice d'accès qui correspond dans la table 1.

La figure 4 montre également que les flux entre les acteurs sont représentatifs du voyage de l'information. Ainsi, pour tenir compte de la transitivité des échanges, nous devons observer les flux complètement. Au lieu d'observer la légitimité d'une interaction entre deux acteurs, nous devons observer la propagation de l'information. Comme le constat du papier (Mao et al. 2009), nous avons présenté la limite du contrôle d'accès et nous allons présenter une autre approche que nous utilisons : l'analyse des flux d'information.

Flux d'information

Dans un système d'exploitation, les flux d'information permettent de contrôler les interactions entre les différentes ressources en tenant compte de la transitivité comme proposé par l'article (Krohn et al. 2007). De cette manière, un service système sensible pourra échanger les informations avec les services autorisés. Ces derniers ne pourront pas repartager ces informa-

tions avec des services non autorisés. Cela implique d'observer tous les flux entre les services. En 2003, le papier (Sabelfeld et Myers 2003) montre l'avancée en 1996 d'un résultat (Volpano, Irvine et Smith 1996) qui réunit les méthodes formelles (Bell et LaPadula 1973) et la description de non-interférence ((Cohen 1977), (Goguen et Meseguer 1982)). Ce papier (Volpano, Irvine et Smith 1996) définit la base du CFI tel que nous l'utilisons dans LIO – notre bibliothèque de CFI.

Propagation de l'information

Dans un langage de programmation, l'information se construit, se propage et s'accède de différentes manières. Illustrons différents flux d'information avec un programme simple : une fonction avec les noms de paramètres non typés suivants `f : (a * b)` $\rightarrow$ `(c * d)`. Cette fonction prends un couple de paramètres (`a` et `b`) et retourne un couple de résultats (`c` et `d`). Cette fonction transforme `a` pour produire `c` et combine les arguments `a` et `b` pour produire `d`. Les arguments peuvent avoir des niveaux de sensibilités différents (par exemple, public et secret). La figure 5 illustre une telle fonction.

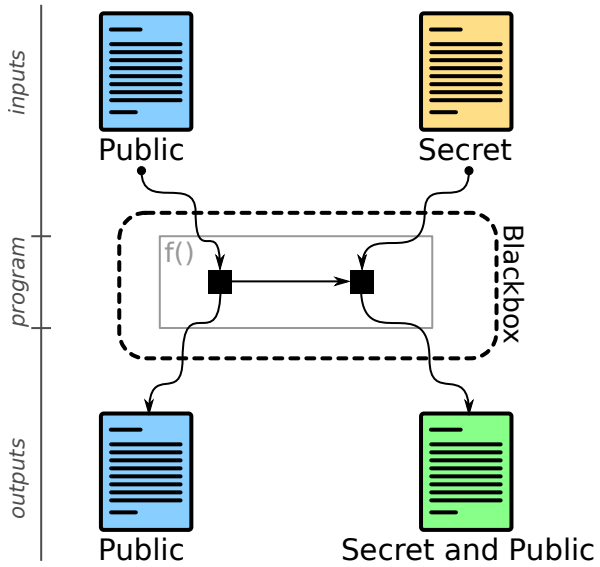

Figure 5 – Flux d'information entre les paramètres et les résultats

L'objectif de la propagation est de connaître la sensibilité à l'origine de `c` et `d`. Si `a` et `b` ont la même sensibilité alors `c` et`d` auront la même également. Toutefois si `a` et `b` ont une sensibilité différente comme illustré sur la figure 5, `d` sera combiné d'une sensibilité publique et secrète. Dans ce cas, `c` aura la sensbilité publique puisqu'il est transformé à partir de l'argument `a`. Nous appelons ce genre de flux "explicite" puisque les données sont liées entre elles.

Il existe de nombreux moyens de lier des informations entre elles. Les flux d'information implicites fabriquent de nouvelles informations non sensibles en fonction d'une information sensible. Le programme ci-dessous permet d'illustrer ce genre de flux. En effet, le retour est une

valeur qui n'est pas construite à partir de la valeur de `s` qui est secrète mais en fonction d'un test conditionnel.

```
if (s == 5) /* secret */
    return true;
else
    return false;
```

Il est également possible d'utiliser le temps pour lier une information. Le programme suivant montre comment fuiter une information selon le temps. Dans cet exemple, le temps d'exécution du programme dépendra de la valeur du secret.

```
if (s == 5) /* secret */
    wait(10);
```

Dans ce dernier cas, nous parlerons de canal caché. Nous différencions un flux implicite d'un canal caché puisque ce dernier sort de la définition du langage. En effet, dans la sémantique du langage C, `wait` est une fonction. En revanche, un flux implicite dans un branchement est propre à la sémantique du langage. Notre travail porte sur la possibilité d'ajouter des contrôles afin d'isoler les informations de niveau de sensibilité dans le cadre des flux d'information implicites dans le langage. Nous ajoutons qu'il existe une étude précise des différents types de flux de l'état de l'art dans l'article (Sabelfeld et Myers 2003).

Traçage de l'information

Comment modéliser la propagation de l'information dans un programme ? Dans les contrôles d'accès que nous avons évoqués plus tôt, nous vérifions juste si un utilisateur peut avoir accès à une ressource donnée et la matrice d'accès permet de modéliser les flux d'un utilisateur vers un autre (Harrison, Ruzzo et Ullman 1976). Ainsi, dans notre exemple de la fonction `f` précédente, nous aurions dû faire une matrice d'accès pour trois sources de flux : *public*, *secret* et *secret et public*.

En 1976, le papier (Denning 1976) propose l'utilisation de treillis pour modéliser les flux d'information. Cette structure algébrique est un ensemble ordonné qui contrairement à la matrice d'accès, permet d'exprimer plus facilement toutes les combinaisons possibles. En effet, dans notre exemple de la fonction `f` précédente, nous devons définir deux niveaux – *public* et *secret* – et trois fonctions : un test et deux opérateurs. Le test vérifie pour deux labels – combinaison de niveaux comme *public et secret* – si l'un est inclus dans l'autre. Par exemple, nous dirons que *public* est inclus dans *public et secret* mais l'inverse n'est pas vrai. Les opérateurs permettent de composer ou décomposer des niveaux entre eux : l'union de *public* et *secret* donne *public et secret* alors que l'intersection entre *public* et *public et secret* donne *public*. Nous rappelons que le treillis borné possède un maximum (*secret et public*) et un minimum ($\emptyset$) qui dans le cadre du

CFI indiquent que le programme a accédé à tout (maximum) ou que le programme n'a accédé à rien (minimum).

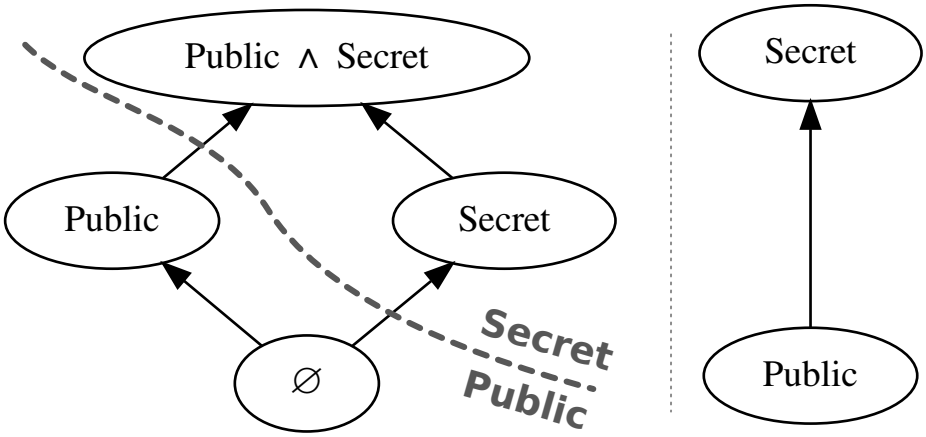

FIGURE 6 – Deux treillis équivalents à deux niveaux : secret et public

Dans la littérature, nous voyons souvent une approche où l'on voit des informations qui sont soit secrètes, soit publiques comme illustré dans la partie *b* de la figure 6. En effet, lorsque l'on mélange une donnée secrète avec des données publiques, le résultat sera au moins secret. C'est pourquoi l'information publique est considérée comme moins sensible que l'information secrète.

Il existe une autre représentation qui garde une trace plus précise. Dans la partie *a* de la figure 6, nous pouvons voir que $\emptyset$ et *public* sont équivalents à l'instar de *secret* et *public et secret*. Les niveaux *public* et *secret* sont distincts, car ils proviennent de sources de sensibilités différentes. Le label vide correspond aux informations qui ne sont ni publiques ni secrètes et mélanger une information publique avec une information secrète donnera une information qui provient de source publique et de source secrète. Comme de nombreux résultats dans l'état de l'art ((MYERS 1999), (STEFAN, RUSSO, MITCHELL et al. 2017),…), nous utiliserons des treillis dans ce document, en recourant ici à la version la plus précise.

Précision du traçage Une fois définie la méthodologie pour assurer la traçabilité des données, il nous est possible de définir la précision. En effet, il existe deux approches : l'approche à grain fin ou l'approche à gros grain. Ces deux approches sont strictement équivalentes en termes d'expressivité (VASSENA et al. 2019), mais elles diffèrent dans leur manière de fonctionner.

Grain fin Cette approche est la plus précise puisque toutes les données sont protégées par le CFI. Cette méthode est utilisée par certains processeurs qui permettent de vérifier directement des programmes entiers sans aucune annotation – seules les entrées/sorties sont annotées (FERRAIUOLO et al. 2018). L'inconvénient de cette approche est le coût du processeur. En effet, un tel

processeur consomme beaucoup plus de mémoire et de puissance puisqu'il faut labelliser toute la mémoire : le CFI est alors appliqué à l'octet près.

C'est l'approche utilisée par les programmes d'analyse statique comme (Myers 1999). En effet, ce programme analyse le code Java où les parties sensibles sont annotées et refuse un programme qui mélange des données secrètes avec une sortie publique. Cette méthode n'a pas de coût supplémentaire à l'exécution puisqu'elle est exécutée uniquement au moment de la vérification du programme – c'est-à-dire à la compilation. Souvent cette approche est associée aux langages de programmation puisque c'est dans le code source que le développeur peut enrichir le plus le code ((Sabelfeld et Myers 2003), (Vassena et al. 2019)).

Gros grain Cette approche est la moins précise puisque seulement un sous-ensemble des données sont protégées. L'idée est d'empêcher des informations partagées entre différents utilisateurs de passer par des flux d'information non voulus. Par exemple, c'est le cas de *Flume*, un outil qui met en place un CFI au niveau des processus (Krohn et al. 2007). Ce qui se passe dans le processus n'est pas observé par le CFI. De fait, le coût si le système est dynamique est bien plus faible puisque le CFI n'est pas actif à chaque octet de la mémoire. Souvent cette approche est utilisée pour les systèmes d'exploitation (Enck et al. 2014).

Toutefois, le problème de contamination des labels devient possible à cause de l'approximation du gros grain. Par exemple, ouvrir un fichier *secret* sans le lire ou perdre l'information pourra rendre le programme *secret*. Nous illustrons cet exemple avec la fonction `labelcreep` ci-dessous :

```
int labelcreep(int secret, int public){
    int r = secret + 2;
    return public * 2;
}
```

Dans le code de `labelcreep`, la valeur du secret est perdue. Bien que dans un CFI à grain fin le retour de la fonction sera de niveau *public*, dans un CFI à gros grain, le retour de la fonction sera de niveau *secret* et *public*. Le papier (Vassena et al. 2019) montre à la fois ce problème et aussi comment le résoudre. En effet, en indiquant les dépendances du retour de la fonction – uniquement *public* dans l'exemple – il est possible de corriger l'approximation.

Mixte Il existe aussi la possibilité de fusionner les deux approches pour obtenir un compromis. C'est exactement ce que fait LIO (Stefan, Russo, Mitchell et al. 2017). L'idée est de permettre de labelliser les entrées et les sorties tout en laissant la possibilité au développeur de labelliser lui-même certaines parties spécifiques de code. Le projet *Laminar* va plus loin puisqu'il permet de labelliser à la fois les processus comme *Flume* mais aussi d'écrire un code CFI (Roy et al. 2009).

Moment du traçage Quand analyser les flux d'information de notre système ? Cette question permet de mettre en lumière deux approches distinctes de vérification : à la compilation ou à l'exécution. Ces deux approches sont opposées puisque l'une se fera une seule fois et l'autre en permanence. Nous appelons ces deux approches respectivement statique et dynamique.

Statique Analyser les flux d'information à la compilation permet de vérifier un nombre assez important de critères puisque le temps de compilation est souvent considéré comme moins important que le temps d'exécution. En effet, une fois le programme vérifié comme étant correct, il est possible de supprimer la notion de CFI. C'est notamment la démarche empruntée par (Myers 1999) qui analyse les programmes à la compilation pour accepter ou non un programme en termes de CFI. Dans la lignée de LIO, il existe une version statique qui utilise l'approche MAC (Russo 2015).

Dynamique Analyser les flux d'information peut également se faire *à la volée*, c'est-à-dire pendant que le programme est exécuté. Cette approche ne permet pas de savoir à l'avance si un programme est correct ou non, mais elle permet de stopper une violation de la politique de sécurité lorsqu'elle est sur le point d'être commise. Cependant, cette surveillance du programme ne vient pas sans un coût. LIO fait partie de ces outils qui utilisent cette approche (Stefan, Russo, Mitchell et al. 2017). De même, cette approche permet de couvrir aussi des programmes inconnus comme dans le cas d'une implémentation matérielle (Ferraiuolo et al. 2018).

Mixte Il existe des tentatives de combiner le meilleur des deux mondes : une vérification statique dans les cas où le coût d'une vérification dynamique est trop important, et une vérification dynamique dans les cas où le programme n'est pas connu à l'avance ou que la preuve d'un tel programme est complexe. C'est notamment l'approche utilisée par Hybrid LIO (HLIO) (Buiras, Vytiniotis et Russo 2015) qui permet de vérifier certains tests statiquement et donc de les omettre à l'exécution.

Après cet aperçu des différentes approches à la fois en termes de précision et en termes de moment de vérification, nous pourrions nous dire que l'approche à grain fin statique est la plus performante – voir la figure 7 – : pas de coût et pas de problème de contamination. Toutefois cela n'est pas complètement vrai : le code n'est pas toujours connu à la compilation. En effet, un programme peut faire appel à des scripts comme le fait un navigateur internet. Le code étant inconnu à la compilation, il est alors normal que les flux ne puissent qu'être uniquement gérés de manière dynamique – par exemple (Austin et Flanagan 2009). De plus, la preuve d'un système peut parfois être plus complexe statiquement que dynamiquement puisque les tests déchargent des obligations de preuve (Buiras, Vytiniotis et Russo 2015) – un test dynamique supprime le besoin de preuve statique.

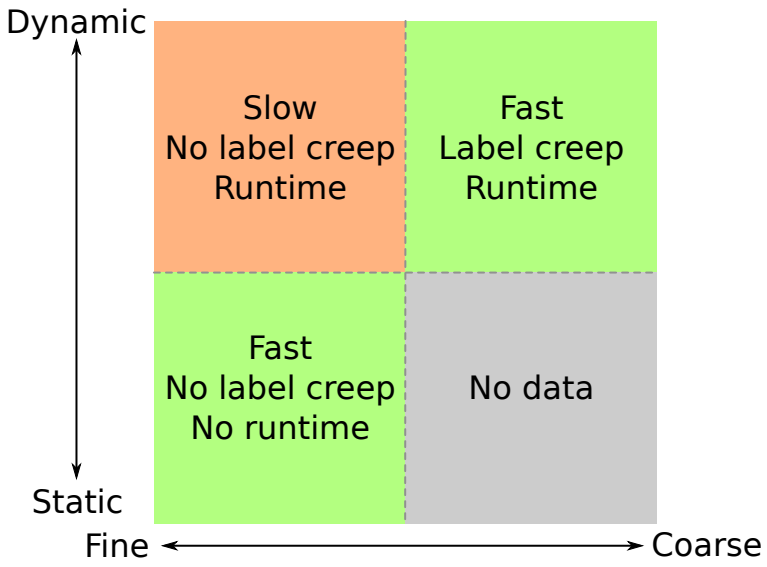

FIGURE 7 – Avantages et inconvénients entre statique/dynamique et fin/gros

Dans la figure 7, nous résumons les avantages et inconvénients entre les trois approches : à grain fin statique, à grain fin dynamique et à gros grain dynamique. L'approche à grain fin dynamique sera la plus lente et la plus coûteuse (*slow*) mais il n'y aura pas de contamination des labels (*no label creep*) et elle pourra supporter du code non connu à la compilation (*runtime*). Par rapport à cette approche, celle à grain fin statique ne pourra pas protéger un programme inconnu à la compilation (*no runtime*). Enfin, l'approche à gros grain dynamique sera un compromis puisqu'elle sera moins coûteuse (*fast*) que sa variante statique et pourra protéger du code inconnu à la compilation (*runtime*), mais elle aura le problème de contamination des labels (*label creep*).

Contrôle de la propagation

L'information se propage et cette propagation peut être mesurée. Ainsi, il nous est possible de déterminer un seuil à partir duquel la propagation dépasse ce que nous voulons. Par exemple, dans notre scénario la propagation est acceptable jusqu'au mélange d'informations provenant d'*Alice* et de *Charles*. Nous pourrions dire que nous ne souhaitons pas que notre système possède une information qui contient un tel mélange.

Cette séparation entre ce qui est acceptable en termes de sécurité et ce qui ne l'est pas s'appelle *la politique de sécurité*. Pouvoir faire respecter une telle politique revient à bloquer la formation d'information non acceptable, ce qui revient à interdire des flux indésirables tels que *Alice* et *Charles*. Lorsque nous parlons de processus, nous dirons que nous isolons le processus *Charles*.

Dans les systèmes de contrôle d'accès, cette politique détermine quels utilisateurs peuvent ou non accéder aux ressources. Dans le cadre du CFI, il s'agit de la même approche. Ainsi, nous pouvons utiliser l'approche MAC dans un CFI pour déterminer si un flux est légitime ou

non (Russo 2015). Plutôt que définir une approche spécifique, LIO définit une *autorisation* qui correspond au plus grand label autorisé (Stefan, Russo, Mitchell et al. 2017).

Concernant l'isolation de programme, le problème de contamination des labels augmente considérablement le nombre de faux positifs dans un CFI à gros grain. Par exemple, si un programme utilise deux entrées, l'une à *Alice* et l'autre à *Charles*, mais retourne une valeur dépendant uniquement de l'entrée d'*Alice*, ce programme sera alors en violation de la politique de sécurité. Cela arrive lorsque le système de labellisation approxime les accès comme c'est le cas dans une approche à gros grain. Pour éviter cette situation, LIO permet d'exécuter une fonction dans un environnement de CFI temporaire pour limiter la propagation (Stefan, Russo, Mitchell et al. 2017).

Non-Interférence

Grâce à la traçabilité de l'information et au contrôle de sa propagation, nous pouvons assurer l'absence de fuite d'information. L'absence de fuite – aussi appelée *non-interférence* – se définit comme l'absence de mélange d'un niveau de sensibilité secret vers un niveau de sensibilité public. Nous dirons alors que les données secrètes n'interfèrent pas avec les données publiques. Par exemple, les données d'*Alice* ne se mélangent pas avec celles de *Charles* même si les données de *Bob* le font. Cependant, comment pouvons-nous nous assurer de manière formelle que notre programme est non-interférent ?

Nous illustrons cette propriété avec deux exemples : le premier est non-interférent et le second ne l'est pas. Dans l'exemple ci-dessous, nous avons deux affichages du message `"ok"` à des sensibilités différentes. Le test conditionnel – de sensibilité publique – peut selon sa valeur interférer dans le niveau public (via `print_public`) ou dans le niveau secret (via `print_secret`). Dans ce dernier cas, il affiche la chaîne de caractères `"ok"` qui dépend d'une information publique. Si le niveau public est inclus dans le niveau secret – par exemple, dans la manière de tracer la propagation de l'information – alors ce programme n'est pas interférent.

```
if (public > 0)
    print_public("ok");
else
    print_secret("ok");
```

Toutefois, comme illustré ci-dessous, si le test conditionnel est de sensibilité secrète, le message affiché dans le niveau public (via `print_public`) est dépendent d'un secret, ce qui, en fonction du traçage de la propagation de l'information, représente un flux non voulu.

```
if (secret > 0)
    print_public("ok");
else
    print_secret("ok");
```

La formalisation de ce *mélange* non voulu est proposée dans le papier (Cohen 1977) en 1977 avec la notion de *dépendance forte* ; ce papier montre comment prouver l'absence de certains flux d'information. De même 5 ans plus tard, le papier (Goguen et Meseguer 1982) montre comment la non-interférence peut aider à vérifier la sécurité d'un programme. En effet, l'interférence y est spécifiée afin de vérifier si un programme respecte une politique de sécurité. Ainsi, le papier (Volpano, Irvine et Smith 1996) conclura en 1996 qu'un programme correctement typé assure la non-interférence. Cette approche – à la croisée entre formalisme du langage et non-interférence – permet d'unifier la non-interférence et un CFI correct.

En 2003, le papier (Sabelfeld et Myers 2003) présente la non-interférence comme l'impossibilité à un attaquant de niveau public de distinguer l'état secret d'un programme. Prenons C un programme ou une fonction. L'exécution de cette fonction $[\![C]\!]$ consiste à relier un état initial vers un état final. La propriété de non-interférence consiste à dire que si les états initiaux (s_1, s_2) sont indistinguables publiquement – nous dirons qu'ils sont publiquement équivalents, $s_1 =_{\text{public}} s_2$ –, les états finaux ($[\![C]\!]s_1$, $[\![C]\!]s_2$) doivent être indistinguables publiquement – nous dirons qu'ils ont un comportement indistinguable, $[\![C]\!]s_1 \approx_{\text{public}} [\![C]\!]s_2$. Nous notons l'équivalence $=_l$ si les valeurs sont identique pour un niveau l et l'observabilité d'un attaquant $\approx_l$ si les valeurs sont indistinguables pour un niveau l. Pour tous états initiaux (s_1, s_2) et niveau de sensibilité (l), cette propriété s'écrit de la manière suivante :

$$\forall s_1, s_2, l.\ \ s_1 =_l s_2 \implies [\![C]\!]s_1 \approx_l [\![C]\!]s_2 \tag{1}$$

Cette formule peut être comprise comme : "*si deux états initiaux partagent les mêmes valeurs publiques, alors les comportements du programme exécuté sur ces états sont indistinguables par l'attaquant*" (Sabelfeld et Myers 2003). Nous notons que dans le papier (Sabelfeld et Myers 2003), la relation d'équivalence $=_{\text{public}}$ et la relation de comportement $\approx_l$ – c'est-à-dire sur les états après l'exécution – sont différenciées. Toutefois, ces deux relations sont identiques dans notre travail.

En 2011, l'article (Stefan, Russo, Mitchell et al. 2011) présente LIO, une bibliothèque CFI en Haskell. Par rapport aux travaux précédents comme (Russo 2015) – qui présente une bibliothèque de CFI qui vérifie statiquement –, cette bibliothèque limite dynamiquement la propagation de l'information avec une autorisation maximale qui représente la politique de sécurité. De plus, ce papier présente une sémantique de LIO ainsi qu'une preuve de sa non-interférence. Cette dernière est intéressante puisqu'elle utilise un lambda-calcul auquel LIO a été ajouté. La correction de la non-interférence de LIO a été prouvée en Coq – nous appelons cette méthode *preuve par simulation* (Pottier et Conchon 2000). Ce travail sera amélioré avec *LWeb* en 2019 (Parker, Vazou et Hicks 2019). Dans cette nouvelle version, la preuve du lambda-calcul est aussi mécanisée par une preuve par simulation mais en utilisant *LiquidHaskell* (Vazou et al. 2014).

Cette méthodologie de preuve – appelée *preuve par simulation* (Pottier et Conchon 2000) –

est différente de l'approche classique de preuve formelle (Hirsch et Cecchetti 2020). En effet, au lieu de prouver formellement la propriété de non-interférence sur la sémantique du langage, cette méthode compare l'exécution d'un programme avec sa version idéalisée. Cette version idéale ne comporte aucune information sensible, il s'agit alors d'une version non-interférente. Dans le cas où les résultats de l'exécution sont équivalents, le programme est non-interférent. Cette preuve porte donc sur l'exécution d'un programme. Comme souligné par le papier (Algehed, Bernardy et Hritcu 2020), la preuve par simulation est réalisée sur un langage jouet – généralement sur un lambda-calcul – qui ne prend pas en compte la manière dont cette bibliothèque de CFI interagit avec le programme. Ce papier (Algehed, Bernardy et Hritcu 2020) utilise la multi-exécution – c'est-à-dire une exécution qui dépend de l'autorisation – pour prouver la non-interférence.

De plus, la plupart des travaux sont réalisés dans des langages fonctionnels comme Haskell. En effet, Haskell permet de contrôler l'*effet de bord* dans un programme en définissant l'impact d'une instruction via les monades. C'est pour cela que de nombreux travaux sur LIO sont implémentés dans ce langage ((Stefan, Russo, Mitchell et al. 2011), (Russo 2015), (Buiras, Vytiniotis et Russo 2015), (Parker, Vazou et Hicks 2019)). Toutefois, ces implémentations ont une approche à gros grain, ce qui implique qu'elles ont un label global flottant – un label courant qui est mis à jour tout au long de l'exécution et qui garde une trace du niveau de sensibilité du programme. Grâce à l'approche monadique ce n'est pas un problème, l'effet est contrôlable. Cependant, de nombreux langages ne proposent pas une telle protection. Ainsi, le papier (Hirsch et Cecchetti 2020) propose une preuve de non-interférence dans un formalisme qui tient compte d'un effet de bord. Cette méthode utilise la théorie des effets pour modéliser un langage avec des effets de bord. Ces travaux déterminent les invariants qu'un langage doit avoir pour prouver la non-interférence.

Forme de labellisation alternative et non-interférence

Dans ce document, nous utilisons des labels basés sur des niveaux de sécurité. Toutefois, il ne s'agit pas de l'unique manière de contrôler la propagation de l'information. Des travaux ont évalué comment quantifier une fuite d'information. En effet, selon la manière dont l'information peut fuir, l'attaquant n'a pas accès à la même information. L'article (Besson, Bielova et Jensen 2013) utilise la quantification de l'information connue pour évaluer la sensibilité d'une variable. Si une variable divulgue trop d'information alors le moniteur peut stopper la transmission de la valeur.

Imaginons un exemple de branchement tiré de l'article (Besson, Bielova et Jensen 2013). Dans cet exemple, une variable globale `browser` contient le nom du navigateur internet de l'utilisateur. Dans un système de CFI comme LIO, un tel code sera rejeté si `browser` est privé et `x` est publique. Cependant, l'article (Besson, Bielova et Jensen 2013) ne rejette pas de code, car il évalue combien d'information peut être divulguée dans un programme.

```
x = 0;
if (strcmp(browser, "firefox") == 0)
    x = 1;
return x;
```

De plus, le résultat dépend de facteurs externes. En effet, si 21% de la population des utilisateurs utilisent un navigateur "*firefox*" alors le résultat de la sortie sera lié à cette proportion. Un attaquant apprendra plus d'information sur 21% de la population et apprendra beaucoup moins d'information sur les 79% restants. Dans une telle situation, la quantité d'information I acquise par un attaquant se note en fonction d'une variable comme illustrée ci-dessous. Nous notons v pour représenter les valeurs possibles de la variable `browser` et $P(\text{browser} = v)$ pour noter la probabilité que cette variable puisse avoir cette valeur.

$$I(\text{browser} = v) = -\log_2 P(\text{browser} = v)$$

Dans le cas de l'exemple du papier (Besson, Bielova et Jensen 2013), nous obtenons 2.25 bits d'information lorsque la variable `browser` vaut "*firefox*" et 0.34 bit sinon. La variable a eu accès à plus d'information dans le premier cas que dans le second cas. Lorsqu'un programme divulgue 0 bit d'information, il est alors non interférent. Cela diffère de l'approche employée dans nos travaux puisque si nous avions utilisé un niveau de sensibilité secrète à la chaine de caractères `"firefox"`, ce programme serait obligé de retourner à *minima* une information de niveau secret. L'approche utilisée dans (Besson, Bielova et Jensen 2013) pour contrôler la propagation de l'information est donc plus flexible. Cet article a également mis en lumière les prérequis pour prouver la justesse d'un moniteur hybride, définit un modèle de moniteur hybride et compare sa précision avec trois autres moniteurs hybrides.

L'article (Besson, Bielova et Jensen 2016) est une extension de l'article (Besson, Bielova et Jensen 2013). Il améliore la précision du moniteur hybride développé précédemment. Pour y parvenir, le raisonnement ne se fait pas uniquement sur des classes d'environnement. Si deux configurations mènent aux mêmes valeurs de sortie, elles sont traitées de manière identique. Voici ci-dessous un exemple tiré de (Besson, Bielova et Jensen 2016) où `h` est secret et les autres valeurs sont publiques :

```
if (h != 0)
    z = x + y;
else
    z = x - y;
return z;
```

Dans une approche comme celle que nous utilisons avec LIO, nous aurions bloqué ce programme. En effet, il existe un jeu de valeurs `x` et `y` qui rend ce programme interférent : `x = 1` et `y = 1`. Dans ce cas, `z` vaudra 0 ou 1 en fonction de `h`. Cependant, toutes les valeurs ne rendent

pas interférent le programme. Par exemple, `x = 1` et `y = 0`. Quelle que soit la valeur de `h`, la sortie vaudra toujours la valeur 1. De plus, l'article (Besson, Bielova et Jensen 2016) a montré que la composition de moniteurs peut être plus précise sans impacter la justesse du contrôle de la propagation.

Questions de recherche

Depuis longtemps la vérification formelle existe, mais l'augmentation de la puissance de calcul et l'avancée de la recherche permettent d'automatiser la vérification formelle. Cela nous autorise à rendre plus réaliste un rêve : la preuve automatique. En effet, la preuve de programme peut être répétitive et donc, pourquoi ne pas utiliser le langage informatique pour apporter des garanties précises avec pour seul effort la rédaction du programme : "*typer, c'est prouver*". Dans notre recherche nous avons essayé de nous rapprocher du modèle "*écrire, vérifier et exécuter*", où l'effort de preuve est réduit au maximum.

Dans l'optique de cette approche, F⋆ permet la vérification d'implémentation cryptographique (Zinzindohoué et al. 2017). Ainsi, une question que nous avons poursuivie est la faisabilité pour la vérification de l'embarqué. En effet, modéliser une architecture est déjà un premier pas dans cette direction (Fromherz et al. 2019), mais nous pouvons appliquer une vérification en tenant compte de l'architecture. En profitant de la génération de code C depuis F⋆ et de la manipulation de registres en C sur l'Arduino, nous posons trois questions de recherche :

1. **Embarqué** : Pouvons-nous modéliser un état interne de la carte Arduino et rejeter tout programme qui ne respecterait pas les contraintes de cet état ?
2. ***Cyberphysique*** : Nous est-il possible d'anticiper l'évolution physique d'un système en fonction du logiciel et rejeter tout programme qui contournerait la spécification ?
3. **Praticabilité** : Quelle est la difficulté d'implémenter un outil système qui correspondrait à l'état de l'art ?

Depuis de nombreuses années, les contrôles de flux d'information permettent de restreindre la propagation des labels selon une politique définie. Nous retrouvons généralement dans les implémentations sur des systèmes complexes une approche à gros grain, qui est moins précise. En effet, l'approche à gros grain approxime les flux d'information, ce qui mène au problème de contamination des labels. LIO est une bibliothèque qui résout ce problème de contamination des labels grâce à un opérateur qui exécute un bloc de code dans une exécution temporaire. Toutefois, cet opérateur ne fonctionne que si le bloc de code n'a pas d'impact sur le système. Si cet invariant est facilement vérifiable en langage Haskell, il ne l'est pas en langage C, qui est un langage utilisé en informatique embarqué et que nous ciblons. Ainsi, est-il possible de modéliser l'impact d'un bloc de code pour obtenir une version extraite en C qui respecte l'invariant de l'exécution temporaire ? La bibliothèque LIO est-elle utilisable dans le monde de l'embarqué ?

De plus, vérifier un programme soulève la problématique du positionnement de la vérification. En effet, nous pouvons vérifier des propriétés à l'exécution – c'est-à-dire sur les données – ou à la compilation – c'est-à-dire sur le code. Nous faisons également remarquer que la vérification sur le code d'un programme est plus facile pour certaines propriétés alors que d'autres le sont beaucoup moins. Généralement, cette différence dépend des connaissances que le programme a sur les données qu'il manipule. Par exemple, un programme qui doit vérifier la justesse d'une structure produite par le programme lui-même est plus simple à vérifier que si cette structure est produite par l'utilisateur. En effet, dans le premier cas, le code qui produit la structure doit assurer la propriété. Dans le second cas, nous devons produire un test qui valide ou non la structure de l'utilisateur. Le premier cas peut être accompli à la compilation, il s'agit d'une vérification statique, tandis que le second cas nécessite une vérification dynamique. Dans nos travaux, nous avons vérifié des propriétés qui peuvent être vérifiées statiquement ou qui peuvent être vérifiées dynamiquement.

Enfin, pour être considérés corrects, les CFI doivent assurer la non-interférence – c'est-à-dire que le système ne permet pas de mélanger des informations sensibles avec des informations publiques. LIO a déjà été prouvée, donc sa définition est correcte. En effet, elle a été prouvée grâce à une méthodologie de preuve appelée preuve par simulation, ce qui consiste à comparer un programme avec sa version idéalisée. Cependant, cette preuve porte sur la bibliothèque et non sur le programme qui l'utilise. Cela permet de dire si la définition est juste ou non. En réfléchissant à cette question, nous avons remarqué que Meta$\star$ est un sous-langage de F$\star$ permettant de réécrire des programmes F$\star$ par métaprogrammation. Cela nous a amenés sur la piste suivante : est-il possible d'utiliser Meta$\star$ pour "idéaliser" un programme puis le comparer avec le programme original en utilisant le calcul de la précondition la plus faible ? Une telle question nous donne l'opportunité d'étudier une automatisation possible de cette méthodologie de preuve. Nous dégageons de ces réflexions, trois questions de recherche :

4. **CFI** : Comment garantir un CFI correct dans un environnement système tel que proposé par la programmation en langage C ?
5. **Mixité d'approches** : Comment avoir un positionnement de la vérification des invariants qui peut être soit dynamique soit statique ?
6. **Automatisation** : Comment automatiser une preuve de la non-interférence d'un système de CFI grâce à la métaprogrammation ?

Tout au long de ce travail, nous avons étudié ces six questions qui nous ont permis de réaliser des contributions que nous décrivons dans la section suivante.

Contributions

Ce manuscrit présente plusieurs contributions. Elles se regroupent en deux catégories : la vérification formelle dans l'informatique embarquée présentée en partie II et la vérification du

CFI présentée en partie III.

Dans le cadre de l'informatique embarquée en partie II, nous présentons trois applications en F⋆ :

— Tout d'abord, nous présentons comment modéliser le fonctionnement d'une Arduino, dans le chapitre 5. Cette contribution permet d'introduire l'aspect *cyberphysique* de la modélisation.

— Ensuite dans le chapitre 6 en section 6.1, nous illustrons une vérification d'ordre *cyber-physique* avec un moteur pas à pas. L'objectif de cette contribution est de montrer que ce genre de vérification est possible dans un langage de vérification de programme.

— Enfin, le chapitre 6 en section 6.2 présente un gestionnaire de mémoire système Memory Management Unit (MMU). Bien qu'elle soit différente des contributions précédentes, cette réalisation permet de vérifier une implémentation de l'état de l'art (Choudhuri et Givargis 2005). De plus, la partie III implémente aussi un MMU.

Ensuite, dans le cadre du CFI en partie III, nous présentons trois travaux de recherche en F⋆ :

— Dans un premier temps, nous implémentons LIO dans le langage F⋆ dans le chapitre 7. Cette implémentation est utilisée dans le chapitre 8. Formellement vérifiée, cette version de LIO étend l'état de l'art en proposant une vérification comme proposée par (Parker, Vazou et Hicks 2019), mais propose également une extraction en langage C.

— Dans un second temps, nous proposons dans le chapitre 9, une combinaison entre la labellisation statique – à la compilation uniquement – et la labellisation dynamique – à l'exécution uniquement. Similaire au travail (Buiras, Vytiniotis et Russo 2015), cette contribution permet de réduire le coût du CFI lorsque la preuve est accessible.

— Dans un troisième temps, nous présentons une tentative d'automatisation de la preuve de non-interférence dans le chapitre 10. Ce résultat limité est le résultat le plus ambitieux : tenter de prouver automatiquement la non-interférence d'une fonction.

Ces contributions permettent d'apporter des éléments de réponses aux différentes questions de recherche.

Plan de lecture du document

Nous avons pu observer dans notre travail qu'une approche système a des subtilités qui rendent complexe la réponse à nos questions de recherche. Nous souhaitons nous rapprocher du modèle "*modélisation, vérification et exécution*". Ce modèle consiste à réduire la différence entre la modélisation et l'application. En F⋆, cela se traduit par une modélisation par le code du programme qui une fois vérifié, s'extrait en C. Nous utilisons ainsi la syntaxe F⋆ comme formalisme principal et nous présentons des exemples extractibles dans le cadre d'utilisation qui est celui de l'informatique embarquée.

Dans cette section, nous présentons les différents chapitres et les liens entre eux. La figure

8 présente graphiquement les interactions de ces chapitres en les regroupant en parties. Chaque dépendance est illustrée par un lien fléché commençant par un point. Pour chaque chapitre, les contributions sont mises en avant par un petit losange.

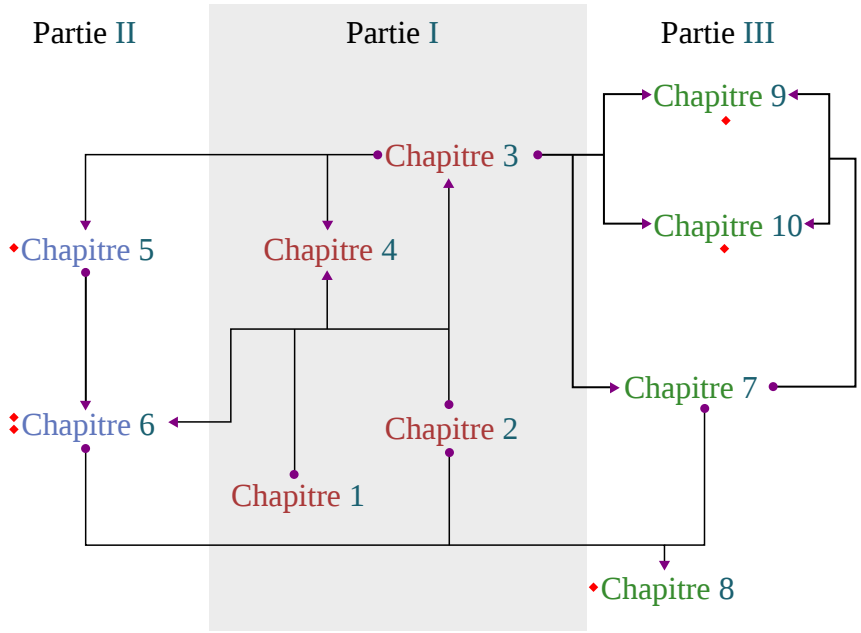

FIGURE 8 – Interactions entre les chapitres

La partie I a pour objectif de présenter les langages utilisés et la manière dont ils peuvent nous servir dans ce travail. Le chapitre 1 présente le langage C qui malgré son ancienneté reste un langage adapté à l'informatique embarquée. Le chapitre 2 présente brièvement F⋆ et son objectif est de donner une initiation au langage. Enfin, le chapitre 3 présente les capacités de modélisation de ce langage. Nous concluons cette partie par le chapitre 4 qui discute des avantages et inconvénients et explique comment interfacer le langage C et F⋆. Un exemple de vérification de pile mémoire est donné pour illustrer pédagogiquement l'interaction des deux langages.

La partie II présente un axe de travail qui approfondit l'utilisation des outils tels que décrits dans le chapitre 4. Cette extension porte sur une modélisation de l'Arduino que nous présentons dans le chapitre 5. Également, le chapitre 6 montre deux applications de ce type de vérification : l'une illustre un moteur – un composant *cyberphysique* – et l'autre illustre un gestionnaire logiciel de mémoire MMU – un composant système.

Enfin, la partie III présente le CFI et aborde comment vérifier de programmes en tenant compte du CFI. Ce travail est fait dans une optique de l'informatique embarquée ou système. Cette optique fait le lien entre les deux parties (II et III) et donne lieu à une implémentation de LIO dans le chapitre 7. Puis dans le chapitre 9, une vérification d'un CFI hybride est présentée afin d'alléger le coût de la preuve, mais également de l'exécution du programme. Enfin, une tentative de l'automatisation de la preuve de non-interférence est présentée dans le chapitre 10.

Nous concluons par un résumé des résultats et une ouverture sur les possibles directions futures de notre travail.

Première partie

Fondations techniques et logiques

Dans cette partie, nous présentons les langages utilisés : le langage C et F⋆. Ces deux langages ne sont pas choisis par hasard et pour permettre au présent document d'être autosuffisant, nous présentons les différentes fonctionnalités de ces langages. Notons que cette partie ne présente pas de contribution scientifique. Son objectif est d'apporter des informations sur les fonctionnalités des langages utilisés dans les exemples de code.

Le chapitre 1 donne une rapide introduction au langage C. Cette introduction couvre la syntaxe, le système de types et termine sur la philosophie de l'utilisation du langage C.

Le chapitre 2 propose une présentation brève de F⋆ à l'instar du chapitre 1. Cette introduction couvre la syntaxe, le système de types ainsi que la notion d'effet. L'objectif de ce chapitre est de donner un aperçu de F⋆ sans rentrer dans les fonctionnalités avancées.

Le chapitre 3 approfondit la présentation de F⋆ en mettant en avant les fonctionnalités avancées. Ce chapitre donne une initiation au calcul de la précondition la plus faible, aux mécaniques de preuves formelles et à la métaprogrammation. Ces fonctionnalités sont utilisées dans les parties II et III.

Enfin, le chapitre 4 donne les raisons des choix de ces langages. Ce chapitre argumente sur les avantages et inconvénients des deux langages. Ce chapitre introduit la raison de la combinaison de ces deux langages : l'extraction depuis F⋆ vers le langage C grâce à KreMLin. Cette dernière fonctionnalité permet de vérifier des programmes système. Une illustration à but pédagogique est présentée en section 4.4 du chapitre 4. Cet exemple est fortement inspiré du tutoriel (Protzenko 2018) mais est adapté pour être plus court.

Pour résumer, la partie I est facultative si le lecteur ou la lectrice est familier avec le langage C, le langage F⋆ et l'extraction via KreMLin. Les chapitres de cette partie peuvent être lus dans n'importe quel ordre. Toutefois, l'auteur du document recommande de lire le chapitre 2 avant le chapitre 3 afin de connaître la syntaxe de F⋆.

Présentation du langage C

Inventé en 1972 par Dennis Ritchie et Ken Tompson dans les laboratoires Bell, le C est un langage de programmation de bas niveau (*C (langage)* 2020). Il a été développé en même temps que le système d'exploitation Unix. Le but de ce langage est d'être proche de l'architecture tout en permettant la portabilité des programmes, ce que la compagnie Bell a réussi avec son système Unix. Le langage C hérite de deux différents langages : BCPL et B. Le B est un langage fortement inspiré du BCPL – par exemple, le seul type du langage est le *mot informatique* – avec beaucoup de fonctionnalités en moins. Ce langage est réduit pour tourner sur des ordinateurs ayant peu de capacités.

Le C n'était ni le premier ni le seul langage de programmation, mais ce qui a fait sa popularité est la portabilité. Les développeurs du langage l'ont conçu pour rendre Unix compilable sur toutes les machines. Avec la première standardisation en 1989 (*The Development of the C Language* 2020), le C est devenu plus largement répandu. Il a été propulsé par le système Unix et cela a continué avec le système Linux.

Code	Description
`v`	Valeur
`e1; e2;`	Séquence
`{ ... }`	Bloc de code
`static type_t var;`	Déclaration de variable globale
`l:`	Définition d'un label
`goto l`	Saut au label `l`
`ret f(params){ ...; return e; }`	Définition de la fonction f
`f(e1)`	Appel de la fonction f
`if (cond) b_true`	Branchement conditionnel
`if (cond) b_true else b_false`	Branchement conditionnel
`switch (e) { case v: e1; break; default: e2; }`	Liste de cas
`for(init; cond; step){ e }`	Boucle pour tout
`while(cond) { e }`	Boucle tant que
`do { e } while (cond);`	Boucle tant que

Table 1.1 : Expression (`e`) en langage C.

Pourquoi le C est-il devenu si populaire ? C'est un langage qui est simple à comprendre, facilement compilable par de nombreux compilateurs et dont la plupart des bibliothèques sys-

tèmes – la GlibC par exemple – sont écrites et maintenues dans ce langage (*The Development of the C Language* 2020). Bien qu'il soit toujours massivement utilisé dans la conception de programmes de niveau système ou embarqués, c'est un langage qui tend à être remplacé dans l'industrie par des langages ayant plus d'abstractions comme le C++, D, C#, Rust ou Java. Les différences entre le C et ces langages pouvant être brièvement résumées au typage et à la sûreté mémoire.

Le langage C est un langage impératif très simple. Par rétro-compatibilité, il accepte la programmation non structurée comme le code *spaghetti* – c'est-à-dire la programmation par sauts dans le code avec des `goto` et des labels qui est la norme en assembleur. Cependant, ce type de programmation est déconseillé, car il expose le développeur à de nombreux risques de bugs. La table 1.1 montre de manière non exhaustive comment une expression peut être formée en C. Nous faisons remarquer que les procédures sont juste des fonctions qui ne retournent rien et les fonctions n'ont aucune contrainte sur les variables globales. De plus, le C possède 3 types de boucles. Nous donnons un exemple de déclaration de variable globale, variables qui peuvent être définies n'importe où dans un programme et qui sont initialisées par défaut à 0. Il est important de comprendre que l'initialisation par défaut ne concerne que les variables globales puisque les variables auront la valeur de la mémoire à leur emplacement, ce qui dépend de plusieurs facteurs comme le système d'exploitation, l'architecture matérielle ou les résidus d'exécution de précédents programmes.

La syntaxe du C étant simple, la définition des types l'est tout autant. Dans la table 1.2, nous listons les principaux types de base – le développeur peut définir de nouveaux types et structures à partir de ces types de base. Les langages B et BCPL n'avaient pas d'autres types que le mot informatique. Le C apporte deux types : le `char` qui désigne les caractères (sur 8 bits) et le `int` qui est utilisé pour les calculs et dont la taille dépend de l'architecture.

Type	Domaine de valeur	Description
`void`	$\emptyset$	Type vide (n'a pas de valeur)
`char`	-128 à 127	Octet
`unsigned char`	0 à 255	Octet non signé
`int32_t`	-2^{31} à 2^{31} - 1	Entiers relatifs (32bits)
`uint32_t`	0 à 2^{32} - 1	Entiers naturels (32bits)
`int16_t`	-2^{16} à 2^{16} - 1	Entiers relatifs (16bits)
`int`	`int32_t` ou `int16_t`	Entiers relatifs
`unsigned int`	`uint32_t` ou `uint16_t`	Entiers naturels
*	`int` ou `NULL` (0)	Pointeurs

TABLE 1.2 : Domaines de valeur des types de base.

Combinés avec le mot-clef `unsigned`, leurs domaines de valeur peuvent être doublés en

perdant l'information du signe. Avec les besoins des développeurs systèmes, le langage C s'est doté de raccourcis qui fixent la taille des nombres afin de pouvoir écrire du code indépendant de la taille variable de `int`. Nous n'avons pas introduit l'intégralité des types comme les flottants ou l'attribut `long` qui permet de doubler la capacité des nombres – passer de 32 bits à 64 bits par exemple. De plus, le C est équipé de nombres flottants comme `float` et `double`. Cependant, dans nos travaux nous n'utiliserons pas ces nombres et donc nous ne les aborderons pas.

Une fois que nous avons vu les principaux types fournis par le langage, nous pouvons parler des pointeurs. Ces derniers sont des adresses mémoires qui pointent sur une structure, un tableau ou une valeur. Les structures et les tableaux sont des données stockées de manière contiguë en mémoire. Ainsi lorsqu'un programme accède au troisième élément d'un tableau, il cherche un élément pointé par l'adresse du tableau incrémentée par deux fois la taille d'un élément.

Code	Description
`type_t t[n] = { 0, ..., n - 1};`	Tableau de `n` éléments de type `type_t`
`type_t  ptr;`	Déclaration de variable
`type_t *ptr;`	Déclaration d'adresse
`t[n] = v;`	Écriture du $n^{\text{ème}}$ élément
`t[n]`	Lecture du $n^{\text{ème}}$ élément
`*ptr`	Déréférencement
`&x`	Référencement

TABLE 1.3 : Tableaux et pointeurs.

L'écriture `t[n]` peut s'écrire ainsi `(t + n)`. Une adresse peut se manipuler comme un entier non signé. Une valeur appelée `NULL` est par convention l'adresse de valeur (0). Elle est utilisée comme valeur d'erreur. Ainsi, manipuler une adresse de valeur `NULL` est considéré comme erroné, alors que toute autre valeur est considérée comme légitime. Cela pose des problèmes pour savoir quelles adresses sont réellement utilisées et celles qui ne le sont pas. La table 1.3 donne un aperçu de l'utilisation des adresses.

Maintenant que nous avons parlé des tableaux et des pointeurs, nous présentons dans la table 1.4 les valeurs qu'une expression peut contenir. Le C accepte des entiers exprimés en système décimal, des entiers en système hexadécimal – ils sont alors préfixés de `0x` – et des entiers en système binaire – alors préfixés en `0b`. Les caractères sont entourés d'apostrophes. Les chaînes de caractères sont entourées de guillemets et sont un raccourci pour désigner l'adresse d'un tableau de caractères constant. Ainsi l'expression `"Bonjour"[3]` retourne `'j'`. Les structures sont similaires à des tableaux mais avec des types différents pour chaque case. L'exemple ci-dessous illustre l'écrasement du champ d'une structure en utilisant des pointeurs :

```
struct data {
    char f1;
```

```
    int  f2;
};
...
struct data a = {'e',5};     // a.f2 == 5
int *p = (int*) &a;
p[1] = 1234;                 // a.f2 == 1234
```

Dans cet exemple, nous déclarons une structure ayant deux membres : un caractère (`f1`) et un entier (`f2`). Nous donnons l'adresse de `a` comme un tableau d'entiers appelé `p` que nous transtypons de `struct data*` vers `int*` et nous écrasons la valeur du second élément avec `p[1]` (les indices de tableaux commencent à 0). Cela marche parce que les champs sont correctement alignés – nous ne détaillerons pas ce mécanisme –, mais cet exemple met en évidence la gestion des données très minimaliste en C ainsi que le manque de protection face à un écrasement.

Code	Description
`x`	Variable
`'a' | "hey" | ... | c`	Constantes
`0 | -1 | 0xcafe | 0b1010 | ...`	Constantes
`struct { char f1; int f2;}`	Structure de données
`(type_t) e`	Transtypage de `e` en expression de `type_t`

TABLE 1.4 : Valeurs (`v`) en langage C.

Pour illustrer la syntaxe du C, nous donnons le code ci-dessous de la copie de deux chaînes de caractères (*C (langage)* 2020). Cette fonction prend deux adresses mémoire sur des tableaux de caractères et pour chaque adresse déréférencée pointée par `d`, la valeur déréférencée pointée par `s` est remplacée par celle de `d`. Les deux adresses sont incrémentées jusqu'à ce que la dernière valeur de `d` soit `0`, ce qui indique la fin de la chaîne de caractères. Si nous inversons `s` et `d`, ce programme ne marche plus puisqu'il ne terminera plus jamais. L'expression `s++` est un raccourci pour l'expression `(s = s + 1)`.

```
void strcpy(char *s, char *d) {
    while (*s++ = *d++);
}
```

Nous faisons remarquer que la boucle n'a pas de corps. C'est normal, car sa condition provoque un effet de bord qui rend le programme terminant. Les exemples de cette section mettent en avant la subtilité du C qui le rend si dangereux à utiliser : tout est possible et sans aucun contrôle de la part du langage. Ainsi, le C s'inscrit totalement dans la philosophie du langage BCPL : "*La philosophie de BCPL n'est pas celle du tyran qui pense savoir ce qu'il y a de mieux*

et qui fait la loi sur ce qui est permis et ce qui ne l'est pas ; BCPL agit plutôt comme un serviteur qui offre ses services au mieux de ses capacités sans se plaindre, même lorsqu'il est confronté à des aberrations flagrantes. Le programmeur est toujours censé savoir ce qu'il fait et n'est pas limité par des restrictions mesquines." (Richards 1969)

Chapitre 2

Présentation syntaxique de F⋆

Le langage de programmation F⋆ a pour objectif de modéliser la spécification et l'implémentation d'un programme. Un atout de ce langage est d'utiliser des types dépendants (Swamy, Hriţcu et al. 2016) et des effets pour représenter l'impact d'une fonction ou d'une séquence d'instructions (Ahman et al. 2017). Grâce à un solveur Satisfiability Modulo Theory (SMT) – par défaut Z3 (Swamy, Chen et Livshits 2013) – il peut vérifier la validité du programme vis-à-vis de sa spécification. Dans le cas où le solveur est incapable de vérifier la satisfiabilité du système, l'utilisateur peut guider le solveur. Cela permet de faire de la preuve de programmes qui une fois vérifiés sont "extraits" – c'est-à-dire transformés – en programmes écrit dans des langages systèmes tels que OCaml, F#, C, WASM et ASM ((Protzenko, Zinzindohoué et al. 2017), (Protzenko, Beurdouche et al. 2019), (Fromherz et al. 2019)).

À quoi sert un tel langage ? Bien que F⋆ soit un langage de recherche toujours en version alpha – c'est-à-dire loin d'être commercialement prêt – il existe de nombreux projets qui démontrent la capacité de F⋆ à être utilisé industriellement. Nous pouvons citer notamment le projet Everest qui développe plusieurs bibliothèques cryptographiques comme miTLS ((*miTLS : A Verified Reference Implementation of TLS* 2020), (Bhargavan et al. 2013)), EverQUIC (*everquic-crypto* 2020) et High-Assurance Cryptographic Library in F⋆ (HACL⋆) (*A High-Assurance Cryptographic Library* 2020). Le projet miTLS est une implémentation de Transport Layer Security (TLS) 1.3 en F⋆ et donc vérifiée. Elle est compilable pour les systèmes d'exploitation Windows et Linux. Le projet EverQUIC est une implémentation du protocole de transport de paquets réseau conçu et implémenté par Jim Roskind à Google (*QUIC* 2020) en 2012. Cette version est implémentée en F⋆ et extraite comme miTLS en C. HACL⋆ est une implémentation d'une bibliothèque cryptographique utilisée dans Mozilla Firefox, la blockchain Tezos, le kernel Windows, le kernel Linux et Wireguard. Ces projets mettent en avant le potentiel industriel de F⋆.

Grâce à ses fonctionnalités théoriques, F⋆ permet d'avoir confiance entre l'implémentation et la spécification (définition des fonctions). Bien que le code puisse être extrait dans des langages système, cette transformation n'est à ce jour pas vérifiée formellement. Nous illustrons avec des exemples concrets dans la section 4.4 comment F⋆ permet de trouver des incohérences facilement. Dans ce chapitre, nous décrivons comment le langage F⋆ est structuré. Bien que ce langage soit proche de F# et OCaml en termes de syntaxe, nous allons voir que F⋆ permet d'exprimer la spécification de programmes grâce à son système de types. Le but de ce chapitre est de fournir une présentation de trois points : la syntaxe, le typage et l'effet de bord.

La présentation de la syntaxe montre comment est structuré un programme. Le typage permettra de mettre en avant l'expressivité de F⋆ tant en termes de types de base qu'en termes de

types définis par l'utilisateur. Cette mise en avant introduit les prédicats, une fonctionnalité clef de F⋆. Enfin, les effets nous permettent de pouvoir modéliser l'effet de bord dans ce langage fonctionnel tel que l'est F⋆.

2.1 Expressions et valeurs

F⋆ est un langage fonctionnel pur, c'est-à-dire qu'il ne permet pas d'effet de bord – l'appel à une fonction avec les mêmes arguments produit toujours le même résultat et les fonctions sont des valeurs de première classe. Cependant, il est possible de modéliser un effet de bord grâce aux effets que nous présentons plus tard.

Code	Description
`v`	Valeur
`e1 e2`	Application
`e t`	Application de type
`i0; i1`	Séquence d'instructions
`if cond then b_true else b_false`	Branchement conditionnel (booléen)
`match e with br1..brn`	Filtrage par motif (liste de cas)
`begin ... end` / `( ... )`	Bloc de code
`let x = e1 in e2`	Définition d'une variable locale
`let rec f = v and f1=v1 ...and fn=vn in e`	Définition mutuellement récursive locale
`e.f1`	Accès à un membre de structure
`let (c:t) = ...`	Constante globale de type `t`
`let f a b = ...`	Fonction globale
`let rec f a b = ... (f a b) ...`	Fonction récursive globale
`let rec f a b = ... and g (c d) = ...`	Fonction récursive globale

TABLE 2.1 : Expressions (`e`) en F⋆.

Les boucles n'existent que par récursivité en F⋆. Le code C ci-dessous illustre la manière d'obtenir en C une boucle **`while`** via une fonction récursive.

```
...
void loop(cond_t cond){      /* while (cond) {   */
    if (cond){
        body;                /*  body;           */
        loop(cond);          /* }                */
    }
}
```

```
loop(cond);
...
```

Dans ce chapitre, nous donnons un exemple de programme – appelé `even` – qui retourne vrai si un nombre naturel est pair et faux sinon. Ce programme possède volontairement des défauts afin d'illustrer les pièges possibles de la programmation ainsi que les mécaniques de vérification. Le code de ce programme en C est le suivant :

```
int even(int x){
    if (x - 2 < 0)
        return !x;  /* x == 0 est équivalent */
    else
        return even(x - 2);
}
```

Le code C ci-dessus est un code récursif, ce qui est considéré comme risqué lors d'une approche système. En effet, ce type de fonction change la pile en fonction de ses arguments, ce qui peut avoir un comportement imprévisible. Pour une entrée positive, la valeur de retour est 0 ou 1 selon que le nombre est pair ou non. Lorsque la valeur 0 est passée en paramètre, le programme retournera bien la valeur 1 pusique 0 est pair. L'opérateur `!` inverse cette valeur puisqu'en langage système la valeur faux est 0 et n'importe quelle autre valeur correspond à la valeur vrai.

En plus de cette subtilité concernant la pile, notre programme n'a pas le même comportement si nous donnons une entrée négative. En effet, dans ce cas le programme retournera toujours faux puisqu'il inversera une valeur différente de 0 qui deviendra 0. Une mauvaise correction sous-optimale (et dangereuse) serait la suivante :

```
int even_positive (int x){
    /* ne prend en compte que les nombres positifs */
    if (x >= 0 && x - 2 < 0)
        return !x;          /* x == 0 est équivalent */
    else
        return even_positive(x - 2);
}
```

Cette nouvelle solution, qui pourtant fonctionne, n'a pas de sens puisqu'elle fonctionne sur le dépassement de capacité des nombres qui font que en dessous d'une valeur spécifique un nombre négatif devient positif. Cette solution nous illustre qu'un programme qui marche n'est pas forcément correct dans tous les cas, ce que nous montrons par la suite.

Par exemple, la fonction `even` peut être écrite en F⋆ de manière très similaire à la version en langage C. Nous verrons plus loin comment F⋆ permet d'enrichir significativement le code afin

de détecter certains problèmes – comme le bon comportement dans les nombres négatifs –, car cette fonction n'est pas sémantiquement correcte. Nous verrons alors que le code de `even` doit être corrigé. Le code ci-dessous est la première implémentation équivalente en F⋆ :

```
let rec even x =
    if x - 2 < 0 then
        x = 0
    else
        even (x - 2)
```

Un point important qui diffère de la programmation système est l'affectation. En effet, F⋆ est un langage fonctionnel pur et les variables ne sont pas modifiables, mais il est possible de les redéfinir. Le code ci-dessous en F⋆ est équivalent au code suivant en C :

```
let x = 8 in
let x = x + 2 in (* x = 10 *)
```

```
int x = 8;
x += 2;      /* x == 10 */
```

La mémoire contient des valeurs qui sont stockées puis modifiées selon les instructions du programme. En F⋆, les variables peuvent contenir une copie d'une autre variable, des constantes, des tuples, des structures ou une abstraction sur des valeurs comme le montre le tableau 2.2 ci-dessous. Pour rappel, l'abstraction est une expression dont les paramètres n'ont pas tous été appliqués.

Code	Description
`x`	Variable
`() | 0 | 1 | true | "hey" | ...`	Constantes
`(v1, v2)`	Tuple
`{[e with] f1=e1;...;fn=en}`	Structure de données [mise à jour partielle]
`λ b →e`	Abstraction sur les valeurs ou types

TABLE 2.2 : Termes sous forme normale (`v`) en F⋆.

L'abstraction – notée λ – est une fonction anonyme qui associe ses paramètres à l'expression qu'elle contient. En programmation système, cela ressemble à une macro cependant elle n'est réduite – c'est-à-dire que les paramètres sont substitués dans l'expression – qu'à l'application avec d'autres expressions.

Maintenant que nous avons vu comment écrire un programme, nous allons montrer comment enrichir sémantiquement ce dernier pour garantir que les fonctions sont utilisées correctement par rapport à ce qui a été conçu.

2.2 Types, prédicats et spécifications

Les valeurs sont fortement typées en F⋆ et le système de type est très riche. Dans la syntaxe de F⋆, il est possible d'exprimer des types dépendants et des types par raffinement. Le tableau 2.3 résume quelques formes utilisées.

Comme il est possible de raisonner sur des types génériques, F⋆ permet aussi de construire des types moins spécifiques. Cela ressemble au polymorphisme mais en étant plus expressif. Par exemple, exprimer le calcul de la taille d'une liste de type quelconque s'écrit en F⋆ en utilisant du polymorphisme de la manière suivante :

Code	Description
α	Variable de type (α:`Type`)
`x:t{phi}`	Type raffiné
`x:t{p x} & y:r{q x y}`	Tuple dépendant
λ `b` $\Rightarrow$ `t`	Type d'une fonction

TABLE 2.3 : Définition de types (`t`).

```
type list α =
    | Nil  : list α
    | Cons : (hd:α) → (tl:list α) → list α

let rec length (l:list α) : nat =
    match l with
        | [] → 0                    (* Nil est noté [] *)
        | hd :: tl → 1 + length tl  (* Cons a b est noté a :: b *)
```

Dans l'exemple de code ci-dessus, nous pouvons voir comment définit un type et typer un code en F⋆. Le type `list` est un type inductif – c'est-à-dire que la structure se référence elle-même dans sa définition – présenté dans le tutoriel F⋆ (TEAM 2020). Cette représentation abstraite permet de raisonner sur son comportement plutôt que sur son implémentation. Par défaut, il existe des types qui sont intégrés dans le langage. Le tableau 2.4 suivant donne les principaux qui sont la base des types raffinés.

Type	Domaine de valeur	Description
`unit`	`()`	Type vide (équivalent de `void` en langage C)
`bool`	`true | false`	Booléen
`Type0`	$\top$ | $\bot$	Booléen (prédicats)
`int`	$-\infty$ .. 0 .. $+\infty$	Entiers relatifs

TABLE 2.4 : Types de base.

Le raffinement est la création d'un type contenant un sous-domaine d'un autre type. Par exemple, imaginons le prédicat `isPrime` qui retourne vrai si un entier passé en paramètre est premier. Nous pourrions alors avoir le type `prime_number` qui correspond à l'ensemble des nombres premiers dans l'ensemble des entiers naturels (`nat`). Sa déclaration serait alors en F⋆ :

```
type prime_number = n:nat{isPrime n}
```

De cette manière, nous pouvons donner des prédicats pour avoir des types plus restreints qui peuvent vérifier des invariants – par exemple un nombre toujours positif. Le code ci-dessous est en partie repris du code source de F⋆ lui-même [1]. Nous donnons l'implémentation des types entiers ainsi qu'un exemple de type borné.

```
type nat = i: int{i ≥ 0}            (* entiers naturels *)
type pos = i: int{i >  0}           (* entiers strictement positifs *)
type nonzero = i: int{i ≠ 0}        (* entiers non négatifs *)

type minutes = n:nat{ n < 60}       (* représentation des minutes *)
```

Les prédicats sont des formules logiques (`phi`) qui sont d'ordre supérieur et donc ils ne sont pas décidables. C'est-à-dire qu'ils ne seront pas traduits en code compréhensible par le processeur puisqu'il n'est pas possible de déterminer si un prédicat est prouvable ou non. Toutefois, ils seront interprétés par le solveur SMT et leur cohérence servira à assurer la cohérence du programme. De fait, un prédicat ou un raffinement disparaît à la compilation. Cela permet de d'exprimer des invariants statiquement.

Code	Description
∀ `b1..bn. p b1 bn`	Pour toutes valeurs des variables `b1...bn`, `p b1 bn` est vrai
∃ `b1..bn. p b1 bn`	Il existe des valeurs pour les variables `b1...bn` tel que `p b1 bn` est vrai
`phi1` ∧ `phi2`	Conjonction
`phi1` ∨ `phi2`	Disjonction
¬`phi`	Négation
`phi1 == phi2`	Égalité de propositionnelle (l'égalité booléenne est notée =)
`phi1` ⟹ `phi2`	Implication
`phi1` ⟺ `phi2`	Équivalence

TABLE 2.5 : Liste des opérateurs pour les prédicats (`phi`).

Le tableau 2.5 donne les opérateurs pour construire des prédicats. Si nous reprenons l'exemple de la fonction `even`, nous pouvons enrichir son typage afin d'exprimer exactement son comportement. Nous savons que la fonction prend un entier quelconque et retourne un nombre positif. Nous définissons un nombre pair comme un nombre divisible par 2 ; ainsi si x est pair alors

1. Le code est disponible dans le code source de F⋆ dans le fichier ulib/prims.fst.

$x \% 2 = 0$.

```
let rec even (x:int) : (r:bool{r ⟺ (x % 2 = 0)}) =
    if (x - 2 < 0) then
        x = 0
    else
        even (x - 2)
```

Le code ci-dessus correspond à la fonction `even` augmentée d'un raffinement qui indique que pour un nombre quelconque (`int`) le résultat `r` est vrai si ce nombre est pair et faux sinon. Cette spécification lie les paramètres et le résultat. Cette fonction ne se comporte pas correctement pour les nombres négatifs. Sans surprise F⋆ va échouer à vérifier ce programme. Le retour de cette fonction est correct pour les nombres positifs mais faux pour les nombres négatifs puisque tous les nombres négatifs seront considérés comme pairs. Normalement, une fois la spécification adoptée et validée, le développeur doit uniquement adapter l'implémentation afin qu'elle soit cohérente avec la spécification. Nous allons cependant prendre des libertés dans cette section pour montrer les interactions de la spécification avec le code. Deux choix s'offrent alors à nous pour faire valider ce code : 1/ changer le type `int` de la variable d'entrée vers `nat` et 2/ modifier le code afin de le valider.

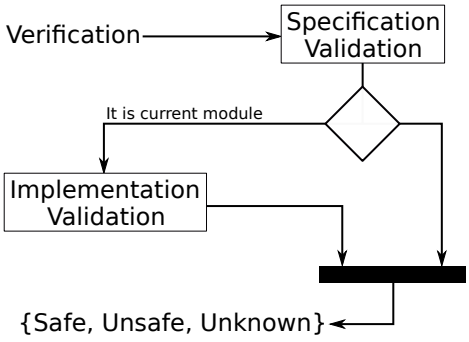

FIGURE 2.1 – Portée de la vérification

Dans la première option, nous pouvons restreindre la fonction `even` aux nombres inférieurs à zéro, ce qui est un peu comme le code de `even_positive` puisque nous n'appliquons le calcul que sur des nombres positifs. En réalité ce code est sémantiquement très différent, car le raffinement disparaît à la compilation. Nous avons parlé d'un dépassement de capacité – problème qui est très connu et qui sera abordé à plusieurs reprises dans ce document – qui n'existe pas dans le code de `even`. En effet, si F⋆ utilise le solveur SMT pour montrer que pour toute entrée dans le programme `even` n'est appelée qu'avec des nombres positifs, alors la fonction `even` sera considérée comme correcte. Cela montre qu'une implémentation imparfaite peut être acceptée si son usage est correct pour toute configuration initiale du programme.

Dans la seconde option, le code respectera pleinement la spécification puisque le code est modifié pour que le résultat soit équivalent à la parité pour tout `int` fourni. Une solution consiste

à transformer un nombre négatif en nombre positif et à changer le type de `even` en `(x:nat) → (r:bool{r ⟺ (x % 2 = 0)})`. Le code de `even4all` sert cet objectif. Nous soulignons que la spécification est la même que celle que nous souhaitons et `even` est appelée correctement par rapport à sa nouvelle spécification. Notons que l'opérateur `%` est définit sur `int`.

```
let rec even (x:nat) : (r:bool{r ⟺ (x % 2 = 0)}) = ...

let even4all (x:int) : (r:bool{r ⟺ (x % 2 = 0)}) =
    if x < 0 then (
        let x':nat = -x in (* aide pour inférer le type *)
        even x'
    ) else
        even x
```

Nous rappelons que ces exemples ne sont utilisés que pour montrer comment utiliser F⋆. Nous avons vu comment exprimer des types enrichis et donner une spécification et nous allons montrer comment formuler une spécification complète puisque F⋆ possède une syntaxe spécifique. Pourquoi F⋆ possède-t-il une syntaxe dédiée à la spécification ? La réponse est simple : la vérification entre la spécification et l'implémentation. L'objectif est de vérifier que le code correspond à la spécification.

La spécification peut être conçue séparément comme une interface à être implémentée. En F⋆, une interface ne contient que la spécification d'un module. Elle est désignée par un fichier suffixé par `.fsti`. Cela permet d'alléger la vérification d'un système complet en découpant la vérification en différents modules qui sont vérifiés séparément. La figure 2.1 illustre ce qui est vérifié selon qu'il s'agit d'une implémentation ou d'une spécification. Lorsqu'un module appelle une fonction d'un autre module, elle est considérée comme exacte. Ainsi, les modules déjà vérifiés ne sont pas revérifiés. Il est intéressant de noter qu'il existe 3 retours possibles après une vérification. En effet, soit le solveur SMT résout le problème (*safe*), soit il trouve une incohérence (*unsafe*) ou soit il ne peut inférer ni l'un ni l'autre (*unknown*).

Pour rédiger une spécification, il suffit d'exprimer le type complet en y changeant `let` en `val`. La spécification peut concerner un code sans implémentation, ce qui permet d'utiliser des entrées/sorties qui sortent du cadre de F⋆ (comme `printf` par exemple). Le rôle du mot-clef `assume` est d'indiquer que le code est extérieur : l'implémentation est hypothétiquement correcte. Cela permet d'inclure des fonctions écrites dans d'autres langages. La table 2.6 montre les éléments de la spécification ainsi que la gestion des modules.

Code	Description
`val x : t`	Déclaration d'une valeur/fonction
`assume val x : t`	Hypothèse sur une valeur/fonction
`module A = M`	Ouverture du module `M` et préfixage en `A`
`open M`	Ouverture du module `M`

`include M`	Inclusion du module `M`

TABLE 2.6 : Syntaxe F⋆

Ainsi si nous reprenons l'exemple de `even` et `even4all`, nous pouvons enlever les types dans l'implémentation et ajouter une spécification pour les deux fonctions comme illustré dans le code ci-dessous :

```
val even     : x:nat → r:bool{r ⟺ (x % 2 = 0)}
val even4all : x:int → r:bool{r ⟺ (x % 2 = 0)}

let rec even x  = ...
let even4all x  = ...
```

À ce point nous avons vu comment enrichir le système de types et par conséquent rendre la sémantique des programmes que nous écrivons plus précise. Cependant, comment F⋆ est-il capable de vérifier un programme ? Nous allons maintenant répondre à cette question.

2.3 Effets

Le principe des effets est simple : enrichir une instruction de son impact dans un système global. Naturellement, les langages fonctionnels purs ne permettent pas l'écriture et la lecture d'un état global. Cela est contraignant les langages fonctionnels utilisent des mécaniques pour autoriser cet état globalisé sous certaines conditions comme Haskell qui utilise la programmation monadique. En F⋆, il est nécessaire de manipuler un état global, d'autant plus que cela facilite la modélisation de programmes non fonctionnels comme ceux écrits en langage C.

En fait, modéliser une variable globale – c'est une pratique courante en programmation système – passe par la modélisation d'un effet. Ce dernier indique comment le système est sera modifié. F⋆ définit alors plusieurs effets qui indiquent comment passer d'une instruction à une autre. Le premier est l'effet `PURE` qui modélise la propagation d'un contexte vide, car nous sommes dans un code fonctionnellement pur. Son argument est une "précondition la plus faible" [2] dont nous détaillons le fonctionnement dans la section 3.1 du chapitre 3.

À partir de cet effet, il est possible de définir de nouveaux effets plus expressifs. Par exemple, l'effet `Pure` est un effet qui dérive de `PURE` et qui prend deux prédicats : une précondition (`pre`) et une postcondition (`post`). Cette définition illustre comment interconnecter ces prédicats pour donner un effet `PURE`. Nous faisons remarquer que `pre` doit être vraie et que pour toutes valeurs de retour de l'expression, `post` doit impliquer `p` qui correspond à la "précondition la plus faible" de la suite du programme. En effet, nous pourrions dire que si `pre` est vraie et que pour tout retour, notre fonction est une hypothèse du reste de notre programme, `post pure_result ⟹ p pure_result`, alors cette fonction ne casse pas les invariants du programme.

2. Le code est disponible dans le code source de F⋆ dans le fichier ulib/prims.fst.

L'idée ici est de définir ce qui doit être vrai avant l'appel de la fonction qui utilise l'effet ainsi que de définir ce qui doit être vrai après. La définition ci-dessous est donc un raccourci qui améliore la lisibilité du code. Ces exemples proviennent majoritairement du code source du langage F⋆ et montrent que les effets peuvent être écrits et paramétrés par le développeur lui-même, ce dernier choisissant l'expressivité de l'effet qu'il implémente.

```
effect Pure (a: Type) (pre: pure_pre) (post: pure_post' a pre) =
    PURE a
        (λ (p: pure_post a) →
            pre
            ∧ (∀ (pure_result: a).
                post pure_result ⟹ p pure_result))
```

L'implémentation de l'effet `Tot` montre que ce type de raccourci peut être moins expressif, car il est possible d'avoir un effet sans argument. Si un code n'a pas besoin de précondition et de postcondition – les opérations pures codées en dur du langage par exemple – il est possible de définir un effet qui n'a aucune condition sur l'instruction courante. Cet effet – appelé `Tot` – s'exprime en F⋆ de la manière suivante :

```
effect Tot (a:Type) = PURE a (λ (p:pure_post a) → ∀ (any:a). p any)
```

Il est également possible de construire des outils de preuves du langage comme `admit` qui admet la suite du programme, `admitP`/`assume` qui admet un prédicat donné et `assert` qui vérifie un prédicat. Cependant, à ce niveau de l'implémentation certains points sont natifs – c'est-à-dire non modifiables sans changer le compilateur – comme l'effet `GHOST` qui signifie que le code disparaîtra à la compilation. Cette disparition n'est pas renseignée par la définition de `GHOST` puisqu'elle est implémentée dans le compilateur. Elle est très utile pour renseigner des informations qui ne sont connues qu'à la vérification du programme.

Nous avons vu ce qu'est un effet et nous allons voir un moyen de les hiérarchiser entre eux. Certains effets peuvent être supérieurs à d'autres, car ils contiennent plus d'invariants que ces derniers. Il n'est donc pas possible de mélanger du code venant de n'importe quels effets sans respecter la hiérarchie. En effet, une information fantôme – sous l'effet `GHOST` – ne doit pas influencer le code même pur. Bien qu'ayant la même définition que `PURE`, `GHOST` est supérieur à `PURE` car les informations peuvent transiter de `PURE` vers `GHOST` mais pas l'inverse. En résumant de manière mémotechnique et amusante : le monde des fantômes de la vérification ne doit pas hanter celui du code vivant compilé. La hiérarchisation se fait par l'*élévation* de l'effet par rapport à un autre grâce à l'opération `sub_effect`. Cette opération prend deux effets séparés par l'opérateur d'élévation ⇝. Le code ci-dessous montre comment l'effet `PURE` est en dessous de `GHOST`. En étendant l'effet `PURE`, l'effet `GHOST` peut contenir des appels à du code `PURE`, mais l'inverse n'est pas possible.

```
total new_effect GHOST = PURE
```

```
sub_effect PURE ⇝ GHOST { lift_wp = purewp_id }
```

Prenons une simple variable globale comme exemple d'illustration. Cet exemple nous permet d'illustrer l'intérêt de la modélisation d'un état global. De plus, nous donnons un exemple concret d'effet dans la section 5.2 du chapitre 5. Dans le code C ci-dessous, la fonction `f` incrémente une variable globale `x`. Cette fonction ne retournera jamais la même valeur deux fois de suite. Nous dirons alors que la fonction a un effet de bord : le fait de l'appeler change le contexte du programme.

```
static int x = 0;
int f(void) {
    return x++;
}
...
int a = f(); // a == 1
int b = f(); // b == 2
```

Nous faisons remarquer que pour modéliser l'impact de `f` sur le contexte du programme – ce que F⋆ permet grâce aux effets – nous allons enrichir une fonction ou un programme avec des préconditions contextuelles. L'idée de cet exemple est de montrer comment modéliser un état global mutable. Bien qu'il existe plusieurs manières de construire un effet, nous utilisons DM4F qui automatise la création de l'effet (Ahman et al. 2017). Nous déclarons alors un nouvel effet avec une valeur numérique comme contexte de la manière suivante :

```
total new_effect GLOBAL = S.STATE_h int
```

Une fois l'effet créé, nous ne pouvons pas utiliser de code dans cet effet puisque cet effet n'est pas hiérarchiquement supérieur ou égal à l'effet `PURE`. C'est pourquoi nous allons *élever* notre effet au-dessus de `PURE`. Nous donnons alors une fonction que nous appelons `lift_pure_global` qui fait la connexion logique entre l'effet `PURE` et l'effet `GLOBAL`. Cette connexion combine des "préconditions les plus faibles", mais nous n'entrons pas dans ces détails dans ce chapitre. Grâce à cette fonction, F⋆ sait que notre effet possède en plus de l'effet `PURE` un entier qui contient une valeur globale – appelé `context`. Le code ci-dessous doit être renseigné pour permettre l'utilisation de notre effet, sinon il ne sera pas possible d'utiliser des opérations `PURE` dans notre effet.

```
unfold let lift_pure_global (a:Type) (wp:pure_wp a) (context:int) (p:GLOBAL?.
    post a)
    = wp (λ a → p (a, context))
sub_effect PURE ⇝ GLOBAL = lift_pure_global
```

Maintenant que nous avons un effet qui étend le langage pur de F⋆, nous allons créer un effet avec une précondition et une postcondition pour rendre la spécification plus simple à écrire.

Cela peut se faire en définissant une forme d'alias qui permet de rendre plus facile la lecture et l'expression de notre effet. En F⋆, cet alias se définit de la manière suivante :

```
effect Global (a:Type) (pre: int → Type0) (post: int → a → int → Type0) =
    GLOBAL a (λ n0 p →
        pre n0
        ∧ (∀ a n1.
            pre n0 ∧ post n0 a n1 ⟹ p (a, n1)))
```

Le code ci-dessus définit un alias de l'effet `GLOBAL`. Cet alias prendra une précondition qui a la valeur de l'entier global *avant* l'appel du code sous l'effet. La postcondition a deux entiers comme paramètres : celui *avant* l'appel et celui *après* l'appel du code dans l'effet. Ainsi pour une valeur quelconque mais fixée de la variable globale, la précondition (`pre n0`) doit être vraie et la postcondition (`post n0 a n1`) doit être vraie pour toutes valeurs possibles de la variable globale et de la valeur de retour de l'expression dans notre effet. La précondition et la postcondition doivent impliquer la suite des préconditions du programme. Notons que le code ci-dessus utilise un type `Type0`. F⋆ permet une hiérarchie de types et le `Type0` correspond au type de `Type`. Plus d'informations sont disponibles sur les univers de types (*Proof-oriented Programming in F* - Universes* 2022).

Par rapport à notre exemple avec l'appel de `f`, nous pouvons maintenant spécifier la fonction avec son effet de bord. Nous savons que la valeur globale est toujours positive puisqu'une valeur statique en C est initialisée à 0. C'est pour cela que nous mettons la précondition `c ≥ 0` avec `c` la valeur globale avant l'appel de `f`. Ainsi tout programme appelant notre fonction sans ce prédicat sera considéré comme faux puisqu'il n'impliquera pas la précondition de `f`. De la même manière, nous donnons un prédicat qui exprime l'effet de bord de notre fonction sur la variable globale : `c1 = c0 + 1`. Ce prédicat exprime simplement que la variable globale *après* l'appel (`c1`) sera incrémentée de 1 (`c0 + 1`) par rapport à celle d'*avant* l'appel (`c0`). Enfin, nous enrichissons la spécification du retour en donnant à F⋆ le lien entre la valeur de retour (`x`) et la variable globale `x = c1`. Le code ci-dessous montre la spécification complète :

```
val f : unit → Global int
    (requires λ c → c ≥ 0)
    (ensures λ c0 x c1 →
        c1 = c0 + 1
        ∧ x = c1)
```

Les mots-clefs `requires` et `ensures` sont définis par F⋆ et servent à améliorer la lisibilité. En réalité, nous pouvons les enlever du code, mais nous les conserverons dans les codes sources que nous présenterons dans ce document pour des raisons de lisibilité et de convention.

Nous pourrions admettre le code de cette fonction comme correct pour que F⋆ vérifie son utilisation. Cette approche peut être utilisée pour des fonctions d'entrées/sorties, mais elle n'est pas intéressante dans notre cas. En effet, elle ne permet pas de vérifier la cohérence de la spécification avec l'implémentation. Souvent, cette dernière doit être très précise et sous-spécifier

une fonction réduit la précision de la preuve : une erreur peut plus facilement exister et ne pas être détectée. Afin de renforcer la vérification par F⋆ l'implémentation doit être liée avec du code, ce qui renforce à la fois la spécification du point de vue du développeur mais aussi pour le solveur. En effet, les invariants inférés depuis le code seront comparés à ceux donnés par la spécification. Ainsi, le code ci-dessous illustre une implémentation de `f` :

```
let f () =
    let x = GLOBAL?.get () in
    let x = x + 1 in
    GLOBAL?.put x;
    x
```

Notre fonction `f` lit la variable globale via `GLOBAL?.get`, incrémente sa valeur, met à jour la variable globale avec la nouvelle valeur via `GLOBAL?.put` et retourne cette même valeur. Les accesseurs/mutateurs appelés respectivement `get` et `put` sont générés par DM4F et peuvent selon la description du contexte être compilables ou juste vérifiables – c'est-à-dire appartenant au monde non extractible. Dans ce dernier cas, les appels à `GLOBAL` disparaîtront, ce que nous verrons dans la partie II. Quelle que soit l'implémentation, F⋆ doit assurer la cohérence entre code et spécification. Nous pouvons alors vérifier un code comme celui ci-dessous :

```
let _ =
    assume (GLOBAL?.get () = 0);
    let a = f() in  // a = 1
    assert(a = 1);
    let b = f() in  // b = 2
    assert(b = 2)
```

En plus d'être un code similaire à l'exemple en langage C, ce code est vérifié et les tests de prédicats (via les `assert`) permettent d'assurer que nous avons exactement ce que nous avons spécifié.

Toutefois, certains peuvent se demander si le code que nous avons donné est vraiment équivalent à notre exemple en langage C. En fait, il ne l'est pas, car en F⋆ le type `int` est infini alors que le type `int` du langage C est fini. En cas de dépassement de capacité la valeur revient à 0 dans un programme C. Cette limite dépend de l'architecture (voir table 1.2). Cependant, il existe des types qui correspondent au `int32_t` du standard C. Cela implique de renommer `int` par le type de la bibliothèque des entiers machines (`FStar.Int32`). Cependant, borner un type ajoute des propriétés à vérifier comme l'absence de dépassement de capacité. Une deuxième option est d'extraire ce code en fournissant une implémentation pour les entiers non bornés et les accesseurs/mutateurs de l'effet `GLOBAL`.

Fondements logiques de F⋆

Dans ce chapitre, nous détaillons le fonctionnement de F⋆ sur une approche plus théorique. Le but est de donner un aperçu des fonctionnalités avancées que F⋆ nous a offertes durant cette recherche. Cet aperçu couvre trois points importants du langage : le calcul de la précondition la plus faible, les preuves et la métaprogrammation. Le premier point est essentiel pour faire de la preuve en F⋆, le second a été utilisé pour effectuer de la preuve de programme embarqué. Enfin, le troisième point, nous a servi à essayer d'automatiser la preuve de programme.

3.1 Précondition la plus faible

Comment vérifier un programme ? Avant de parler du calcul de la précondition la plus faible, nous devons parler de ce qu'est la logique de Hoare. Cette logique est un formalisme ayant pour objectif de montrer que des programmes informatiques sont corrects par rapport à une spécification (Hoare 1969). Le principe de ce formalisme consiste à ajouter à un programme une précondition et une postcondition. La précondition P est considérée vraie au début de l'exécution du programme C. La postcondition R doit être vraie à la fin de l'exécution du programme C. Si P est vraie alors R est vraie après exécution de C si ce dernier termine. Ces trois éléments forment un triplet de Hoare noté de la façon suivante :

$$\{P\}C\{R\}$$

Ce formalisme ne permet pas de raisonner sur l'état, mais sur l'évolution. En fait, P implique R. Toutefois, il est supposé que C termine : pour que R soit validé, C doit avoir terminé. Si la logique de Hoare est d'une justesse partielle – appelée également correction partielle : la postcondition ne sera vraie qu'une fois que le programme termine. En revanche, si la logique de Hoare est d'une justesse totale : pour toute précondition validée, la postcondition sera toujours vraie. Cette approche implique de prouver la terminaison du programme puisque exécuter C implique être dans un état qui valide R si l'état initial satisfait P.

Pour illustrer cette différence entre justesse partielle et totale, prenons un exemple de programme ci-dessous. Ce programme effectue la multiplication de deux paramètres dont l'un est modifié.

```
int f(int x, int y){
    return y*(x+1);
}
```

Si nous choisissons comme postcondition $R : f(x, y) = 0$, il est possible alors de fournir plusieurs préconditions valides d'un point de vue de la logique de Hoare :

1. $P : x = -1$
2. $P : y = 0$
3. $P : x = -1 \wedge y = 0$
4. $P : x = -1 \vee y = 0$

Cependant, toutes ne se valent pas. En effet, la troisième est beaucoup plus forte que la quatrième. Et les deux premières sont plus fortes que la quatrième. De manière simplifiée, plus une précondition est faible, plus elle est minimale et générique.

Là où la logique de Hoare ne permet que de vérifier la cohérence d'un système, le calcul de la précondition la plus faible (réciproquement le calcul de la postcondition la plus forte) permet d'assurer que la précondition est minimale. Ainsi, ce calcul permet de manière déterministe de retrouver la quatrième précondition comme réponse minimale par rapport à la postcondition donnée ($R : f(x, y) = 0$).

Il est nécessaire de noter que la terminaison du programme doit être prouvée dans le cadre du calcul de la précondition la plus faible. Dans le cas des boucles, cela pose un problème qui peut être abordé de deux manières différentes : la justesse partielle ou la justesse totale. La première consiste à vérifier que l'invariant de boucle est toujours correct et la seconde montre également que la boucle termine (*Predicate transformer semantics* 2020).

Ce calcul est décrit par Edsger Dijkstra dans l'article (Dijkstra 1975) et permet d'ajouter un prédicat à un programme. L'idée est de construire automatiquement des termes sous forme de triplet de Hoare et donc de réduire la charge de travail. Cette construction se fait au travers de transformeurs de prédicats. Ces derniers indiquent comment construire depuis les règles du langage ou de l'arithmétique, les préconditions.

Dans le cas du calcul de la précondition la plus faible, le programme sera lu depuis la fin vers le début comme illustré par la table 3.1. Grâce aux transformateurs de prédicats, le développeur doit apporter la preuve que $x = -1$ ou que $y = 0$. Dans le cas contraire, la précondition $\top$ n'est pas suffisante pour garantir la postcondition $f(x, y) = 0$.

Code	Ordre	Précondition
`int f(int x, int y){`	3	$x = -1 \vee y = 0 \iff f(x, y) = 0$
`  return y*(x+1)`	2	$y * (x + 1) = 0$ possède deux solutions : $x = -1$ et $y = 0$
`}`	1	$f(x, y) = 0$

Table 3.1 : Illustration du calcul de la précondition la plus faible sur un programme.

Lorsque F⋆ analyse un programme, il utilise les prédicats d'une part fournis par le dévelop-

peur et d'autre part construits par le calcul de précondition la plus faible. Si une incohérence apparaît, cela stoppe la vérification. Dans notre exemple de programme en table 3.1, le développeur doit supposer la précondition la plus faible à f, s'il souhaite prouver la postcondition.

Dans le code ci-dessous, nous illustrons l'aspect calculatoire et l'aspect définition dans F⋆. En effet, ce code F⋆ décrit deux fonctions : `f0` et `f1` – `f1_corrected` est une version corrigée de `f1`. La première fonction n'est pas spécifiée, alors que la seconde l'est. En rédigeant le lemme `proof_of_f0`, le calcul de la précondition la plus faible est déroulé, ce qui revient à écrire $x = -1 \vee y = 0 \implies (x+1) * y = 0$. Ce prédicat est correct, ce qui permet de valider le lemme. Toutefois, la fonction `f1` échoue à la vérification. En effet, le développeur ajoute la postcondition $f1(x, y) = 0$ via le raffinement. Une fois le calcul de la précondition la plus faible déroulé, nous avons $\top \implies (x+1) * y = 0$ ce qui n'est pas vrai pour toutes les valeurs de x et y – par exemple la solution $x = 1$ et $y = 1$. Ainsi, le développeur doit rajouter la précondition la plus faible $x = -1 \vee y = 0$, ce que nous pouvons voir dans la fonction `f1_corrected`.

```
let f0(x y:int) : int = (x+1) * y

let proof_of_f0 (x y:int) : Lemma
    (requires x = -1 ∨ y = 0)
    (ensures f0 x y = 0)
= ()

[@expect_failure]
let f1(x y:int) : r:int{r = 0} = (x+1) * y

let f1_corrected(x:int) (y:int{x = -1 ∨ y = 0}) : r:int{r = 0} = (x+1) * y
```

Dans le cas de F⋆, le solveur SMT est utilisé pour résoudre les contraintes. Il s'agit d'obligation de preuves : les prédicats demandent à ce que F⋆ ou le développeur prouvent que le programme les satisfait.

Code	Domaine de valeur
`let rec even x =`	$x \in]-\infty, +\infty[$
`  if (x - 2 < 0) then`	$x \in]-\infty, +\infty[\rightsquigarrow x \in]-\infty, 2[$
`    x = 0`	$x \in]-\infty, 2[\wedge x \,\%\, 2 = 0$
`  else`	$x \in]-\infty, +\infty[\rightsquigarrow x \in [2, +\infty, [$
`    even (x - 2)`	$x \in [2, +\infty[\wedge [2, x-2[\ll [2, x[$

TABLE 3.2 : Code avec des invariants.

Reprenons l'exemple de `even` où la spécification est incorrecte ; `val even : x:int →r: bool{r ⟺ (x % 2 = 0)}`. Nous illustrons le code de la fonction avec des préconditions sim-

plifiées dans la table 3.2. Ces préconditions ne concernent que l'ensemble de valeurs de `x`. Pour cet exemple, nous utilisons comme notation $\ll$ pour ordonner deux domaines et $\rightsquigarrow$ pour indiquer que le domaine de `x` est réduit vers un autre domaine. Ces préconditions répondent à deux questions : pourquoi F⋆ ne prouve pas la postcondition et pourquoi la fonction récursive termine.

Nous remarquons que les préconditions soulèvent un problème puisque sachant le domaine de valeur de `x` – soit le domaine $]-\infty, 2[$ –, la proposition $x \ \% \ 2 = 0$ ne peut pas être correcte. En effet, il existe un contre-exemple : -3. F⋆ échoue alors à la vérification puisqu'il peut montrer qu'un contre-exemple est possible. Ainsi nous répondons à la première question.

Concernant la seconde question, nous remarquons à l'appel récursif de la fonction que les intervalles diminuent avec l'appel suivant. En effet, lors de l'appel à `even (x - 2)` l'espace de `x` se réduit de l'intervalle initial $]-\infty, +\infty[$ à $]2, +\infty[$. Comme la fonction `even` s'arrête pour une valeur de `x` inférieure à 2 la fonction ne pourra être récursive que pour une valeur supérieure à 2. Comme l'intervalle $[2, x-2[$ est plus petit que l'intervalle $[2, x[$, nous pouvons dire que la fonction converge et donc qu'elle termine. En effet, lors du premier appel de `even` la valeur de `x` est dans l'intervalle $[2, x[$ et lors du second appel de `even` la valeur de `x` est dans l'intervalle $[2, x-2[$. Nous pouvons également enrichir la spécification de `even` en précisant comment le domaine de valeur se réduit de la manière suivante :

```
val even : x:nat → Tot (r:bool{r ⟺ (x % 2 = 0)}) (decreases %[x;x-2])
```

Cette spécification utilise la clause `decreases` qui est un paramètre de l'effet `Tot`. Nous expliquerons le fonctionnement des effets dans la suite et ici nous illustrons comment indiquer un ordre lexical pour aider F⋆ à prouver la terminaison de la fonction `even`.

Si nous avions implémenté la fonction `even_positive` – vue en section 2 du chapitre 2.1, le domaine de valeur aurait été $]-\infty, 2[$ pour le cas d'arrêt – l'ensemble au moment où la fonction s'arrête – qui ne décroît pas avec le temps. En définitive la fonction `even_positive` ne marche que parce qu'un dépassement de capacité a lieu. Ce qui marche en pratique, mais pas en théorie.

Lorsque nous appelons un code F⋆, nous pouvons utiliser des prédicats qui sont souvent de la forme d'une précondition et d'une postcondition. Dans le cas de `even`, la précondition est `x` $\geq$ `0` et la postcondition est `r` $\iff$ `(x % 2 = 0)`. En effet, les raffinements sont implicitement des préconditions et des postconditions. Cette forme ainsi posée ressemble aux triplets de Hoare $\{P\}C\{Q\}$ avec $P : x \geq 0$ la précondition, $Q : r \iff x\%2 = 0$ la postcondition et $C : even(x)$ un programme. Cependant, comme nous l'avons décrit précédemment, le concept de précondition la plus faible est plus précis : les triplets de Hoare sont une relation entre P, Q et C.

Dans le cas de la précondition la plus faible, P doit être la plus faible précondition sur C qui produit un état et Q est une postcondition qui satisfait cet état. Nous notons $wp(C, Q)$. Ainsi, la fonction wp transforme le prédicat Q avec l'état produit par C en une précondition la plus faible qui est vraie pour C et Q. De cette façon, F⋆ utilise ce principe pour valider l'implémentation, car le code C produit un état qui doit satisfaire la postcondition. Lorsque nous parlons de pré- et postcondition, il s'agit de deux prédicats : la précondition P doit impliquer $wp(C, Q)$ qui est

la précondition la plus faible de C et de la postcondition Q. De cette manière, F⋆ vérifiera que la précondition implique la précondition la plus faible calculée par F⋆.

Par abus de langage, nous utiliserons les préconditions et les postconditions comme des triplets de Hoare. C'est ainsi que nous utilisons F⋆ et ce dernier en calculant la précondition la plus faible va être en mesure de valider ou non une spécification que nous lui donnons. De plus, le calcul de la précondition la plus faible peut exprimer les triplets de Hoare : $\{wp(C, Q)\}C\{Q\}$. Nous illustrons le concept avec un exemple où le programme produit une addition.

```
let example (a:int) : Pure (int)
    (requires pre)
    (ensures λ x → x > 1)
=
    a + 1
```

Dans le code ci-dessus, nous avons une précondition qui doit valider la postcondition appliquée à l'état produit par le code de la fonction – ici `x` est le résultat de retour de la fonction. Un exemple de précondition valide est `a > 10`. Cependant, la précondition la plus faible est `a > 0`. Nous avons vu comment F⋆ vérifie la cohérence d'un programme.

3.2 Preuves

Le langage F⋆ est suffisamment expressif pour que ses termes transportent un prédicat. Cela permet de construire une formule appelée la "précondition la plus faible" – que nous présentons en §3.1 – et d'avoir une représentation logique du programme. Cette approche permet de voir les termes comme des prédicats et de raisonner dessus. F⋆ s'appuie alors sur un solveur SMT pour résoudre le prédicat résultant et donc inférer si le programme est cohérent ou non.

Lorsque nous avons voulu vérifier la fonction `even`, nous avons utilisé le raffinement pour vérifier que le résultat – un nombre est pair ou non – est correct pour toute entrée. Nous avons alors montré que notre fonction retourne vrai ou faux en fonction de la parité du paramètre de la fonction `even`. Cela nous a permis d'exhiber des problèmes notamment dans le cas des nombres négatifs. Cette approche est appelée vérification *intrinsèque* puisqu'elle est liée au type.

Cependant, il est possible de prouver des propriétés d'une manière externe. Cette approche est appelée vérification *extrinsèque*. Pour cela F⋆ validera une fonction qui construit un terme correct et cette construction pourra être alors validée par le solveur SMT. Cela permettra de rejouer la fonction qui agit comme lemme – c'est-à-dire un résultat démontrable, souvent intermédiaire d'une preuve complète qui est le programme vérifié.

Nous prendrons un simple exercice d'écolier pour illustrer ce fonctionnement de F⋆. Le but de cet exemple est d'expliquer comment un terme est construit et comment guider le solveur SMT pour le valider. Nous allons montrer que si un carré est pair alors sa racine l'est également. En mathématique nous écrivons $E(x^2) \implies E(x)$ avec E le prédicat qui évalue la parité

d'un nombre. Afin d'éviter une confusion avec les exemples précédents, nous utilisons dans cet exemple de nouveaux noms. En F⋆, nous commençons par définir les fonctions de parité `aeven` et d'imparité `aodd`. Cela nous permet de définir les types raffinés pair `teven` et impair `todd`.

```
let aeven (x:int) : Tot (r:bool{r ⟺ (x % 2 = 0)}) = x % 2 = 0
let aodd  (x:int) : Tot (r:bool{r ⟺ (x % 2 ≠ 0)}) = x % 2 ≠ 0

type todd  = x:int{ aodd x}
type teven = x:int{aeven x}
```

Pourquoi utilisons-nous le raffinement ? En réalité le raffinement apporte des hypothèses sur les valeurs : ces prédicats restreignent l'ensemble des valeurs d'un type. Ainsi, le lemme `odd_sum_gives_even` utilise deux paramètres raffinés, contrairement à la définition du lemme `even_sum_gives_even`. Ces deux approches sont équivalentes. Ces deux lemmes avec le lemme `mixed_sum_gives_odd` décrivent les propriétés de sommes entre les nombres pairs et impairs. Nous faisons remarquer que nous avons un lemme (`even_and_odd_relation`) qui vérifie que la parité est l'inverse de l'imparité.

```
let odd_sum_gives_even (a b:todd) : Lemma (ensures aeven (a + b)) = ()

let even_sum_gives_even (a b:int) : Lemma
    (requires aeven a ∧ aeven b)
    (ensures aeven (a + b)) = ()

let mixed_sum_gives_odd (a:teven) (b:todd) : Lemma (ensures aodd (a + b)) = ()

let even_and_odd_relation (x:int) : Lemma (ensures aeven x ⟺ ¬(aodd x)) =
    ()
```

Dans le code ci-dessus, tous les lemmes sont vides, ce qui est normal puisqu'un lemme retourne `()` et que le SMT est capable d'inférer le raffinement de retour – c'est à dire l'obligation de preuve du lemme. Il arrive cependant que ce dernier ait besoin d'être guidé. Notre exemple d'exercice est souvent résolu par contraposée – c'est-à-dire par la preuve de l'inverse du problème. Nous inversons alors la proposition et nous avons la formule $I(x) \implies I(x^2)$ avec I la fonction d'imparité. Nous exprimons une telle formule en F⋆ ci-dessous :

```
let odd_implies_odd_square (a:int) : Lemma
    (requires aodd a)
    (ensures aodd (a * a))
=
    let k = a / 2 in
    assert(a = 2 * k + 1);
    assert(a * a = (4 * (k * k)) + (4 * k) + 1)
```

Dans cette implémentation, le lemme contient des indications pour le solveur SMT. En fait, nous décomposons le problème en aidant le solveur SMT pour avancer dans la résolution. Pour le SMT, un nombre impair est valide sous la forme $2k + 1$ et nous lui donnons l'intuition dont il a besoin pour montrer l'imparité du carré du nombre donné en paramètre $(4k^2 + 4k + 1)$. Maintenant nous pouvons prouver notre exercice de la manière suivante :

```
let even_square_implies_even (a:int) : Lemma
    (requires  (aeven (a * a)))
    (ensures (aeven a))
=
    if aodd a then
        odd_implies_odd_square a      (* application de la preuve *)
    else
        ()
```

Nous faisons observer que nous avons deux cas possibles capturés par un branchement ainsi qu'un appel au lemme `odd_implies_odd_square`. Cela s'explique simplement : soit le nombre est impair et dans ce cas nous appelons la contraposée qui prouve notre problème, soit le nombre passé en paramètre est déjà pair et nous n'avons rien d'autre à prouver. Cependant, à quoi cela peut-il servir de prouver extrinsèquement une telle propriété ?

La réponse à cette question peut être illustrée avec l'exemple suivant : imaginons une fonction `work` qui prend un nombre pair et effectue un calcul quelconque. Même si F⋆ sait que le paramètre de la fonction a pour carré un nombre pair – illustré ci-dessous par la fonction `f` – il ne sera pas capable de prouver la cohérence du programme. Afin de montrer cela nous appliquons la preuve en appelant le lemme `even_square_implies_even` que nous avons précédemment présenté.

```
assume val work : teven → Tot (bool)

let f (a:int{aeven (a * a)}) =
    even_square_implies_even a;      (* application de la preuve *)
    work a
```

Nous avons montré dans cette section comment réaliser une preuve principalement en utilisant le solveur SMT. Il existe également un système de tactiques qui permet d'aller beaucoup plus loin mais que nous ne présenterons pas. Cette interconnexion entre le code d'un programme et la preuve est une solution élégante pour prouver des propriétés qui transcendent le code informatique (voir le chapitre 6). Cependant, le système de preuve n'est pas le seul outil que F⋆ nous offre pour raisonner sur le comportement des programmes.

3.3 Métaprogrammation

La métaprogrammation permet de programmer non pas avec des données mais avec des programmes. Autrement dit, au lieu de développer un programme, il s'agit de développer un générateur de programme. Ainsi, cette section décrit comment programmer des programmes en F⋆ via un sous-ensemble du langage appelé Meta⋆ (Martínez et al. 2019). En effet, ce sous-ensemble permet d'écrire des tactiques – fonctions qui étendent le langage. Sans rentrer dans les détails techniques, il existe des constructions en Meta⋆ comme les classes de types – appelées *typeclass* – qui permettent du polymorphisme *ad-hoc*. Ce genre de construction – généralement écrite en dur au niveau du compilateur – est très intéressant pour étendre l'expressivité ou pré-calculer des résultats afin d'optimiser le programme.

Si nous reprenons l'exemple de la fonction `even` avant d'avoir corrigé les problèmes avec `even4all`, nous pouvons écrire deux types de fonctions : celle que nous avons (`original`) et celle que nous voulons (`wanted`). Nous pouvons définir ces types de fonction comme le code ci-dessous et nous donnons aussi la signature de la fonction `even` telle qu'elle est définie dans la section 2.1.

```
type original = (x:nat) → Tot (r:bool{r ⟺ (x % 2 = 0)})
type wanted   = (x:int) → Tot (r:bool{r ⟺ (x % 2 = 0)})

val even : original
```

Notons une remarque importante : nous n'utilisons pas les types machines mais cela est un détail pratique qui peut facilement être adapté aux nombres machines. Également, la vérification a vocation à trouver des problèmes dans le code – elle n'a pas vocation à corriger automatiquement un algorithme. Nous montrons uniquement comment "modifier" un programme dans cette section. Dans la présentation du langage nous avons montré comment utiliser le raffinement pour trouver des cas – les nombres négatifs – qui retournent une valeur erronée. Nous allons montrer comment modifier notre fonction `even` avec la métaprogrammation.

Ce que nous souhaitons est d'avoir une fonction équivalente en termes de comportement à la fonction `perfect_even` décrite ci-dessous et qui est une implémentation exacte de la fonction qui nous sert d'illustration jusqu'ici.

```
val perfect_even : wanted
let perfect_even x = x % 2 = 0
```

Nous écrivons alors une fonction qui transforme une fonction de type `original` vers une fonction de type `wanted`. Pour y arriver nous utilisons Meta⋆ qui nous permet de fabriquer un nouveau terme comme l'illustre le code ci-dessous :

```
let makeWanted (f:original) : wanted =
    let x : wanted = λ (x:int) → f (if x < 0 then -x else x)  in
    normalize_term (synth_by_tactic (λ () → exact (quote (x))))
```

La fonction `makeWanted` permet d'encapsuler l'appel de `f` pour effectuer la correction. Nous avons une fonction qui prend un naturel et qui retourne s'il est pair ou non. Comme nous retournons la valeur absolue du paramètre `x`, cette nouvelle fonction est équivalente à la fonction `even4all`. La fonction `synth_by_tactic` prend une tactique – fonction qui construit et manipule des termes F⋆ – et retourne un terme que nous normalisons avec la fonction `normalize_term`.

Nous vérifions alors l'équivalence entre la fonction `perfect_even` et la fonction `even` transformée – nous l'appelons `corrected_even`. Cette vérification sera très simple pour F⋆ puisque nous avons un raffinement précis et un exemple simple. En réalité, sur des exemples plus complexes F⋆ nécessite d'être guidé et d'avoir accès à beaucoup plus de ressources.

```
let _ =
    let corrected_even : wanted = (makeWanted even) in
    assert(∀ x. corrected_even x = perfect_even x)
```

Dans cette section, nous avons vu comment transformer des termes en F⋆. Nous présenterons un exemple concret de l'utilisation de la métaprogrammation pour vérifier la non-interférence de programmes dans le chapitre 10.

Chapitre 4

Choix et raisons des langages

Dans ce chapitre, nous présentons les motivations qui nous ont poussés à utiliser F⋆. Dans un premier temps, nous donnons les raisons d'utiliser ou ne pas utiliser le langage C ou F⋆. Dans un second temps, nous expliquons en quoi la combinaison de ces deux langages nous intéresse. En effet, F⋆ s'extrait en C, ce qui permet de bénéficier des avantages des deux langages. Enfin, nous illustrons cette combinaison par un tutoriel qui montre comment vérifier une implémentation puis l'extraire. Cette approche qui permet de passer de la vérification à un exécutable est utilisée dans tout le reste de ce document.

4.1 Intérêts et limites du langage C

Nous avons vu la syntaxe du langage C ainsi que certains de ses pièges syntaxiques. D'après la philosophie du BCPL, seul le développeur devrait être tenu responsable de l'échec d'un programme. Ainsi, lorsqu'un programme est d'une criticité cruciale, est-il judicieux de faire confiance uniquement au développeur ? Nous montrons dans les parties II et III qu'utiliser des restrictions peut devenir vital pour assurer des éléments de confiance dans un programme critique. Toutefois, est-ce la seule raison pour laquelle une telle mesure soit intéressante ? Finalement, est-ce que la syntaxe est le seul piège du langage C ?

En réalité non, puisque le langage C peut avoir des comportements indéfinis. L'article (Hathhorn, Ellison et Roşu 2015) cite le standard C où ce type de comportement est considéré comme résultant de l'utilisation d'une construction non portable ou erronée d'un programme ou d'une donnée sans que cette dernière soit restreinte par le standard. Nous pouvons illustrer l'exemple d'un comportement indéfini avec la fonction de la racine carrée inverse rapide ((*Racine carrée inverse rapide* 2020), (Lomont 2003)). Cette fonction s'appuie sur des conversions de type flottant vers entier ainsi que des manipulations bit à bit des valeurs. Elle passe d'une représentation flottante pour la modifier manuellement et avec une heuristique arriver à estimer la valeur de la racine carrée inverse. Cette fonction est plus rapide que son équivalent matériel, mais moins précise : elle marche dans la grande majorité des cas, ce qui ne pose pas de problème dans son cadre d'utilisation. Cet exemple, qui fut très utilisé dans le monde du jeu vidéo, montre comment les comportements indéfinis sont utilisés pour optimiser les calculs avec des contreparties – ici au détriment de la précision.

Il est important de faire remarquer que les compilateurs tirent également avantage de ces comportements indéfinis. Cependant, bien qu'utiles dans des situations très précises, les comportements indéfinis sont extrêmement difficiles à connaître et occasionnent de nombreuses

erreurs dans le monde industriel (Hathhorn, Ellison et Roşu 2015). Pour éviter ce genre de situation, le développeur doit connaître précisément les postulats du compilateur (et de la machine parfois) pour les utiliser correctement. C'est pourquoi il existe des travaux pour modéliser ces portions de code dangereux et pour refuser un programme qui les utilise ((Hathhorn, Ellison et Roşu 2015), (Stewart et al. 2015)) – ainsi le projet CompCert (Stewart et al. 2015) est un compilateur qui rejette tout programme faisant appel à des comportements indéfinis.

Mais pourquoi utiliser un langage aussi dangereux ? Le C est un langage extrêmement simple, portable et proche de l'architecture matérielle. Il n'apporte aucune surcouche comme un moniteur d'exécution – logiciel qui contrôle le bon fonctionnement de certaines abstractions d'un langage – pour apporter une protection logicielle. Ainsi, ce langage reste le langage le plus facile pour développer au niveau du système d'exploitation et de l'embarqué. Il est donc intéressant de vérifier un programme puis de le traduire dans le langage C afin d'obtenir un programme ayant des propriétés spécifiques. Nous verrons comment assurer ces propriétés dans la partie II.

4.2 Intérêts et limites du langage F⋆

Le choix du langage F⋆ pour écrire des programmes n'est pas un choix sans inconvénient. En l'état actuel du développement de F⋆, il existe des freins majeurs à une utilisation industrielle. En effet, c'est un langage de recherche toujours en développement et de nombreuses fonctionnalités sont encore incomplètes, ce qui est un frein temporaire, mais qui devrait, selon l'évolution de ce langage, s'améliorer avec le temps. Nous pouvons citer des problèmes de syntaxe ou d'extraction qui nous ont posé des difficultés lors des réalisations techniques de nos contributions. Par exemple, nous pouvons mentionner la complexité de la rédaction de la spécification qui engendre un coût important et non négligeable. En effet, écrire dans le langage F⋆ nécessite d'avoir une spécification riche et détaillée. Toutefois, cette difficulté est compensée par la possibilité de vérifier le programme de manière automatique dans un grand nombre de cas. Ainsi, le code ne pourra pas contenir d'erreurs logiques ou d'inattention. Une dernière limitation importante est le langage lui-même qui implique de réécrire le code historique pour le vérifier, mais cette dernière est compréhensible puisque l'expressivité du langage F⋆ est bien supérieure à celle du langage C comme nous l'avons présenté dans cette partie I.

Cependant, F⋆ a quand même un gros avantage : celui d'être un outil de vérification qui intègre du code, s'extrait en C et surtout utilise un solveur SMT pour réduire le travail de preuve. Cela permet d'obtenir un langage qui pourrait techniquement supporter un projet complet comme un système d'exploitation. De fait, F⋆ s'inscrit dans les attentes du monde industriel où l'effort de preuve est considéré comme un coût. Ainsi, à l'avenir et avec l'automatisation de la preuve formelle, les entreprises qui conçoivent du code critique ne seront peut-être pas les seules à utiliser des outils de preuve formelle.

4.3 Vérification et extraction

Maintenant que nous avons vu comment exprimer un programme en F⋆, nous allons montrer comment F⋆ est utilisé dans un projet pour générer du code vérifié. Le langage permet d'avoir du code vérifié au sein d'un programme comme un système d'exploitation, un pilote ou une bibliothèque logicielle. Il est important de comprendre que tout le code ne peut pas être vérifié : la modélisation a une portée plus ou moins grande selon les limites des outils de vérification ou du code déjà existant. Comme illustré dans la figure 4.1, nous définissons trois parties importantes : la partie sûre qui est vérifiée de l'implémentation à la spécification, une partie non sûre qui contient du code non vérifié et une partie tampon qui définit comment la partie sûre communique avec la partie non sûre. La partie tampon assure par sa spécification que la partie sûre communique avec la partie non sûre.

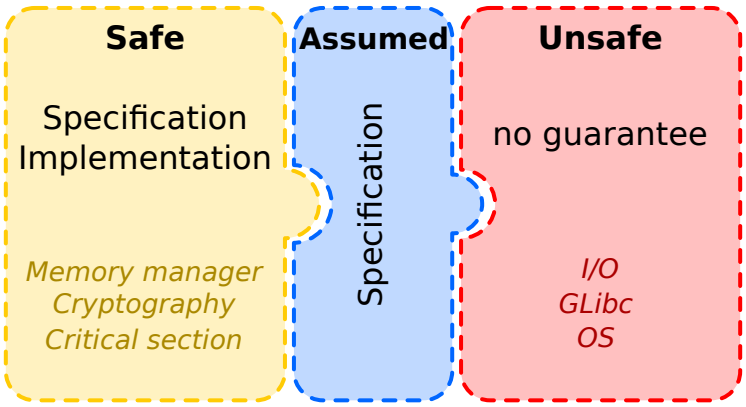

FIGURE 4.1 – Confinement entre le code sûr et non sûr

Ce confinement entre le code sûr et non sûr est important puisque le code non sûr peut casser tous les invariants. Imaginons que nous exposons – c'est-à-dire rendons accessible – la fonction `even : nat → bool` à un code non sûr. Sachant que le raffinement disparaît à la compilation, rien n'empêche ce code non sûr de l'appeler avec des entiers négatifs. Cela réduit à néant les efforts de vérification. Par contre, nous pouvons exposer uniquement `even4all : int →bool` ce qui permet à un code non sûr de respecter la spécification pour tout l'ensemble possible du type `int`.

Dans un projet, le code F⋆ est vu comme une sous-partie du programme qui est vérifiée et qui respecte un certain nombre d'invariants, par exemple une bibliothèque cryptographique (ZINZINDOHOUÉ et al. 2017). C'est pourquoi F⋆ est conçu pour s'intégrer facilement dans un projet informatique. Ainsi le code écrit en F⋆ n'a pas vocation à rester dans ce langage mais à être extrait dans des langages couramment utilisés dans l'industrie comme le langage C. La figure 4.2 montre les différentes étapes qui permettent de passer d'un programme à un autre.

Pour extraire un programme en C, F⋆ va utiliser un solveur SMT – il s'agit de Z3 dans la figure 4.2 – pour vérifier le programme. À ce moment le compilateur applique des transformations pour enlever des informations qui n'ont aucun sens en langage C, comme le code qui sert

à la vérification seulement ou encore le raffinement qui n'existe pas en langage C. À la fin de multiples transformations, le code est exprimé dans un langage intermédiaire appelé Low⋆. Ce dernier est un sous-ensemble du langage F⋆ (P. Wang et al. p. d.).

Nous tenons à faire remarquer que Low⋆ n'est pas L⋆. Le premier est un langage intermédiaire alors que le second est un sous-ensemble de F⋆ avec des bibliothèques qui permettent d'utiliser pleinement les fonctionnalités du langage C. En effet, L⋆ est une modélisation de la mémoire telle qu'elle fonctionne en C (P. Wang et al. p. d.). Nous ajoutons qu'il existe un tutoriel qui montre comment s'utilise cette bibliothèque (Protzenko 2018). L'avantage de L⋆ est de donner une modélisation fidèle et traduisible du C. Dans ce processus de traduction, il existe un langage intermédiaire appelé Low⋆.

Une fois le code transformé en Low⋆, il est envoyé à KreMLin – un traducteur de code Low⋆ vers le langage C. C'est pendant cette traduction que les éléments de L⋆ seront traduits en éléments du langage C (`I32.t` traduit en `int32_t`). Cette traduction est pour l'instant codée en dur dans KreMLin et il n'est pas possible de la changer sans toucher le code du compilateur. Ce compilateur va traduire le code Low⋆ en C⋆ : un sous-ensemble déterministe du langage C qui se rapproche de Clight dans CompCert. C'est ce code final qui sera retourné comme du code C.

Une fois le code généré en C, les fichiers peuvent être compilés avec différents compilateurs en fonction des besoins du projet. Il est important de noter que contrairement à CompCert (Leroy 2009), de nombreux compilateurs ne sont pas vérifiés et peuvent introduire des erreurs à la compilation. Toutefois, pour des raisons techniques nous serons amenés dans les parties II et III à utiliser Gnu C Compiler (GCC) puisque c'est le seul compilateur qui compile sur notre architecture cible.

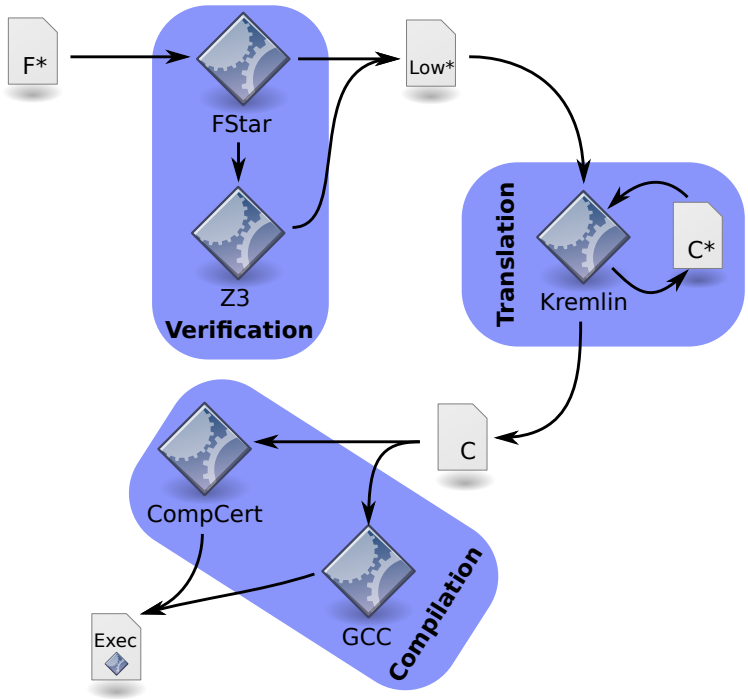

Figure 4.2 – Étapes entre la compilation et la vérification

Avant d'illustrer l'extraction par un exemple, nous allons faire une distinction entre deux termes qui peuvent porter à confusion : *compilation* et *traduction*. Ces deux termes sont sémantiquement similaires puisque la compilation est une transformation d'un langage source vers un langage cible, mais pour le reste du document nous allons utiliser le terme de traduction lorsque nous appliquons des transformations dans des langages similaires – l'extraction F⋆ vers Low⋆ par exemple. En effet, lorsque GCC compile du C vers de l'assembleur, il change radicalement la structure du programme en adaptant l'organisation des données selon le modèle mémoire de l'architecture ou du système d'exploitation : le programme est structurellement adapté, ce que nous appellerons *compilation*. Cependant, lorsque F⋆ et KreMLin appliquent leur transformation, ils ne changent pas la structure mais adaptent la syntaxe pour être conforme au langage C, ce que nous appellerons *traduction*. Ce genre de transformation est utilisé par GCC qui se sert du programme AS pour traduire l'assembleur lisible par un être humain en code binaire lisible par un processeur.

Maintenant, illustrons l'extraction avec notre fonction `even`. Nous attirons l'attention sur l'utilisation des types infinis qui n'ont pas d'équivalent en langage C. C'est pour cela que nous devons adapter notre code avec des types machines – c'est-à-dire utiliser des éléments de L⋆ – qui ont une correspondance en C. Il existe plusieurs modules qui décrivent de tels nombres – par exemple `FStar.UInt32` et `FStar.Int32` qui décrivent respectivement les types `uint32_t` et `int32_t`. Nous réécrivons le code de `even` en conséquence :

```
module U32 = FStar.UInt32

val even : (x:U32.t) → Tot (r:bool{r ⟺ (U32.v x % 2 = 0)})
    (decreases %[U32.v x; U32.v x - 2])
let rec even x =
    if U32.(x `lt` 2ul) then
        x = 0ul
    else
        even U32.(x `sub` 2ul)
```

Nous raffinons les types en utilisant des réflexions (via la fonction `U32.v`) qui permettent de travailler non pas sur des termes de types machines mais en termes de nombres naturels (`nat`). Cette transformation – appelée réflexion – vers des types plus abstraits est plus naturelle pour F⋆ et le solveur SMT. Ainsi, on pourra reconnaître l'ordre lexical donné pour indiquer la terminaison de `even`. La notation `U32.( ... )` permet d'inclure les fonctions du module `U32` dans le bloc de code. Maintenant regardons le code C généré ci-dessous :

```
bool even(uint32_t x) {
    if (x < (uint32_t)2U)
        return x == (uint32_t)0U;
    else
```

```
        return even(x - (uint32_t)2U);
}
```

Comme nous pouvons le voir, les deux codes sont structurellement identiques et le code résultant est lisible par un être humain. Nous complétons cet exemple en ajoutant la fonction `even4all`.Toutefois, nous rencontrons une difficulté dans cette fonction. À l'heure où nous écrivons ce document, l'opérateur unaire *moins* n'est plus opérationnel. Cela nous permet de montrer comment inclure du code externe que nous allons supposer sûr. Imaginons une fonction `inv` dont nous savons qu'elle respecte parfaitement sa spécification ; c'est-à-dire qu'elle inverse le signe d'un nombre signé. Nous incluons cette spécification, en indiquant à F⋆ que cette fonction sera liée plus tard. Nous avons ci-dessous une implémentation de `even4all` :

```
module I32 = FStar.Int32

assume val inv : (a:I32.t) → Tot (b:I32.t{I32.v b = -I32.v a})

val even4all : (x:I32.t) → Tot (r:bool{r ⟺ (I32.v x % 2 = 0)})
let even4all x  =
    if I32.(x `lt` 0l) then (
        let x':U32.t = Cast.int32_to_uint32 (inv x)  in
        assert(U32.v x' = I32.v (inv x));
        even x'
    ) else
        even (Cast.int32_to_uint32 x)
```

Nous souhaitons faire remarquer la proximité du code F⋆ et de celui utilisant L⋆. Ainsi à l'instar de `U32.v` qui réfléchit le type en `nat`, la fonction `I32.v` réfléchit en `int`. Il est important de noter le test du prédicat (via `assert`). Bien que ce test soit inutile, il illustrera l'effacement des informations relatives à la vérification. Le code C généré pour cette fonction est le suivant :

```
bool even4all(int32_t x) {
    if (x < (int32_t)0) {
        uint32_t x_ = (uint32_t)inv(x);
        return even(x_);
    } else
        return even((uint32_t)x);
}
```

En plus d'être une traduction, le code perd une information qui n'a pas de sens à l'exécution : les prédicats. L'appel à la fonction `inv` est toujours présent et elle peut être implémentée dans un fichier externe au système de vérification de F⋆. Cette solution est bien sûr une illustration des fonctions externes, car ces fonctions sortent complètement du cadre de la vérification et elles

peuvent casser les invariants de notre code. Par exemple, `inv` pourrait être l'identité – fonction retournant la même valeur que celle passée en paramètre – et tout ce que F⋆ a tenté de prouver serait validé mais faux en pratique. Pour éviter cela, nous donnerions le code de la fonction comme ci-dessous :

```
val inv : (a:I32.t{I32.v a > -(pow2 31)}) → Tot (b:I32.t{v b = -v a})
let inv a =
    let x:I32.t = (0l `I32.sub` a) in
    x
```

Remarquons que le paramètre de `inv` est plus précis, ce qui nécessite d'adapter la signature de `even4all`. Ce changement est important puisqu'il permet d'éliminer un cas qui échoue en C à cause du complément à 2. En effet, un nombre dans un ordinateur varie de -2^n à $2^n - 1$ ce qui implique que -2^n (`-(pow2 31)`) n'a pas d'équivalent positif. Ainsi, la vérification rejette une implémentation qui a un tel cas particulier.

Dans cette section, nous avons présenté comment F⋆ est en mesure de vérifier un programme et l'extraire en langage C. Nous avons vu comment lier ce code avec un code externe et comment raffiner les types afin de vérifier l'implémentation. Cependant, F⋆ est beaucoup plus puissant que ça puisqu'il permet de prouver des propriétés, ce que nous présentons dans la section suivante avec un exemple.

4.4 Exemple de vérification d'une pile mémoire

De nos jours, les développeurs n'ont pas ou ont peu à se soucier de la gestion de la mémoire. Bien que ces tâches soient gérées par la plupart des langages ou des systèmes d'exploitation, elles doivent être implémentées et vérifiées. Dans certains cas comme dans l'embarqué, le développeur ne peut pas se permettre le luxe d'utiliser une gestion automatique de la mémoire qui apporte un surcoût et une complexité supplémentaire. Dans d'autres cas, c'est l'aspect de criticité qui entre en jeu. Nous devons nous assurer qu'un programme fasse correctement ce qu'il est supposé faire.

En effet, un nombre conséquent d'attaques sur les systèmes d'exploitation modernes ont pour objectif un gain de privilège. Lorsqu'un attaquant peut modifier la pile système, il peut changer de manière arbitraire le cours d'exécution ou même des valeurs spécifiques. Considérons le programme ci-dessous exprimé dans un pseudo-code proche de l'assembleur comme exemple :

```
1 ; memory declaration
2 integer     varA    42
3 stack_t     stack   03
4 integer     varB    14
5 ; code start
6 pop         ; returns 42
```

```
7  push    55  ;
8  push    00  ;
9  push    01  ;
10 push    02  ;
11 push    99  ;
```

Ce programme est syntaxiquement correct puisque que toutes ses instructions sont complètement valides. Cependant, il est incohérent sur le plan de la sémantique. En effet, si nous regardons la mémoire avant le démarrage de l'exécution (à la ligne 5), elle contient les valeurs telles que présentées dans la partie A de la figure 4.3. La partie B de la figure montre, quant à elle, l'état après l'exécution du programme. Nous voyons que les variables globales `varA` et `varB` ont été modifiées. La structure en mémoire `stack` dépasse et écrase la mémoire voisine.

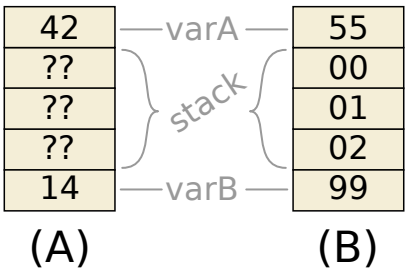

FIGURE 4.3 – Dépassement de mémoire et pile mémoire

L'utilisation d'une telle structure implique d'avoir une excellente connaissance de son fonctionnement, de son implémentation ainsi que de l'environnement où elle est exécutée. Plus un système devient complexe et plus il deviendra dangereux d'utiliser de telles structures. Nous montrerons dans cet exemple comment vérifier une pile mémoire et ainsi assurer la cohérence de sa sémantique, ce qui empêche d'exprimer un programme ayant un dépassement de pile.

En effet, une pile mémoire mal utilisée peut avoir des conséquences extrêmement graves. Si dans un système, cette faille permet la fuite de donnée, elle peut devenir mortelle si le composant qui échoue est un composant critique. C'est ce type d'erreur qui est supposé être à l'origine de nombreux accidents sur certains modèles de voitures Toyota. L'origine exacte n'est pas connue mais beaucoup de questions subsistent quant à la qualité du code qui gère le système d'accélération. En effet, dans certaines conditions l'ordinateur de bord écrasait un bit en mémoire – la cause pourrait être liée à un dépassement de pile – ce qui a pour effet d'accélérer la voiture violemment. Nous ajoutons qu'il est possible de trouver plus d'informations dans les articles (KOOPMAN 2020) et (CUMMINGS 2016).

Cet évènement indésirable nous rappelle à quel point un code simple peut être vital. Le but de cette section est de montrer comment F⋆ peut être utilisé pour contraindre le développement de logiciel afin d'éviter tout code non cohérent ou au comportement indéterminé.

4.4.1 Objectifs de cet exemple

La pile mémoire est une des structures de données les plus importantes puisque utilisée dans chaque processeur et dans chaque programme. Nous considérons dans cet exemple une pile de taille fixe qui peut seulement empiler et dépiler. On comprend aisément l'enjeu de la vérification de son bon fonctionnement. En effet, que se passe-t-il quand l'action de dépilement est utilisée sur une pile vide ? De même, que se passe-t-il lorsque l'action d'empiler est effectuée sur une pile pleine ?

Si nous pouvons donner une réponse aux questions précédentes, il n'existe pas de réponse universelle, car chaque développeur peut implémenter une pile différemment en fonction de la situation, ce qui a conduit à appeler ces réponses des comportements "effet indésirable" : il s'agit d'un comportement supplémentaire par rapport à ce que le programme est supposé faire. Cela montre qu'un programme ne doit pas sortir du cadre qui lui est fixé. Nous montrerons comment faire pour empêcher ces situations mais nous nous pencherons également sur une approche permettant de supprimer le code qui contrôle le bon fonctionnement de notre pile dans certains cas particuliers. Cet exemple est fortement inspiré du tutoriel en ligne de L⋆ (Protzenko 2018) qui montre l'implémentation d'une mémoire tampon circulaire.

4.4.2 Définition fonctionnelle

Comme illustré dans la figure 4.3, nous décrirons une pile de taille fixée. Elle doit donc posséder 3 informations : la taille maximale, le nombre de données valides actuellement ainsi que les données stockées. Comme le langage C utilise beaucoup les pointeurs, nous les utiliserons et nous devons prendre en compte le risque que la structure de données puisse évoluer dans le temps. Bien que nous omettrons pour l'instant les problèmes liés à la concurrence, nous devons faire attention à l'impact des fonctions sur la structure. L'objectif est par exemple d'éviter une fonction qui enlève un élément de la pile sans mettre à jour son nombre d'éléments. Nous devons être sûrs que les pointeurs manipulés pointent toujours sur des données allouées. Dans le cas contraire, nous pourrions avoir une pile structurellement valide mais incohérente.

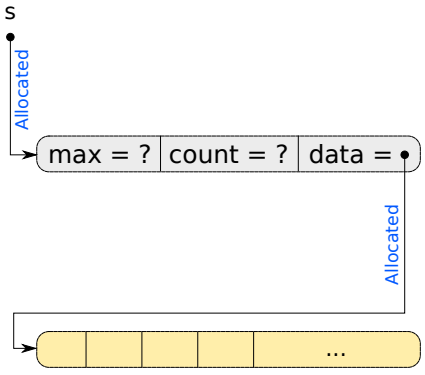

Figure 4.4 – Structure de la pile mémoire

À cause du modèle mémoire en C, nous devons gérer manuellement ces informations. Dans la figure 4.4, nous décrivons la structure de données, que nous représentons avec un pointeur appelé *s* qui décrit les champs correspondant aux 3 informations précédemment décrites. Le champ *data* est un pointeur sur un tableau de données dont la taille est d'au plus *max* éléments. Les deux autres champs sont des entiers qui décrivent le nombre d'éléments et le nombre maximum d'éléments.

Avec cette description, nous pouvons déduire les propriétés que notre pile mémoire doit respecter à tout instant. Pour une pile de type α, nous pouvons donner une définition en raffinant les champs de la manière suivante en F$\star$:

```
type stack (α:eqtype) = {
    max   : U32.t { U32.v max   > 0 };
    count : U32.t { U32.v count ≥ 0 ∧ U32.v count < U32.v max };
    data  : buffer α { length data = U32.v max };
}
```

Le code ci-dessus force par construction les propriétés sur les champs. Il ne sera pas possible de construire une pile invalide et la pile ne pourra contenir que des valeurs comparables (`eqtype`). Pendant la durée de vie de notre pile, nous devons vérifier la cohérence des pointeurs. Nous pouvons donc écrire le prédicat logique suivant :

```
let wellformed (#a:eqtype) (h:mem) (s:pointer (stack a)) : GTot Type0 =
    live h s
    ∧ (let s0 = deref h s in
        live h s0.data
        ∧ modifies (loc_disjoint
                (loc_buffer s)
                (loc_buffer s0.data)) h0 h1
    )
```

Le but du prédicat `wellformed` est de vérifier si les pointeurs sont valides dans une mémoire donnée. Le prédicat `live` vérifie que le pointeur donné est alloué dans la mémoire `h`. De façon similaire, le prédicat `loc_disjoint` vérifie que les pointeurs ne se chevauchent pas. La macro `deref` permet juste de récupérer les valeurs dans la mémoire `h`.

Maintenant que nous pouvons construire correctement une pile mémoire, nous allons montrer comment la modifier. En effet, l'utilisation des pointeurs permet de travailler sur des données mutables, ce qui doit être modélisé. Ce changement de donnée se fait par le biais de deux opérations : `push` pour empiler et `pop` pour dépiler. Le code suivant montre un programme qui génère les états présentés en figure 4.5.

```
1 ; initialization     count = 0   max = 6
2 push 42       ;      count = 1   max = 6
3 push 0xba     ;      count = 2   max = 6
4 pop           ;      count = 1   max = 6
```

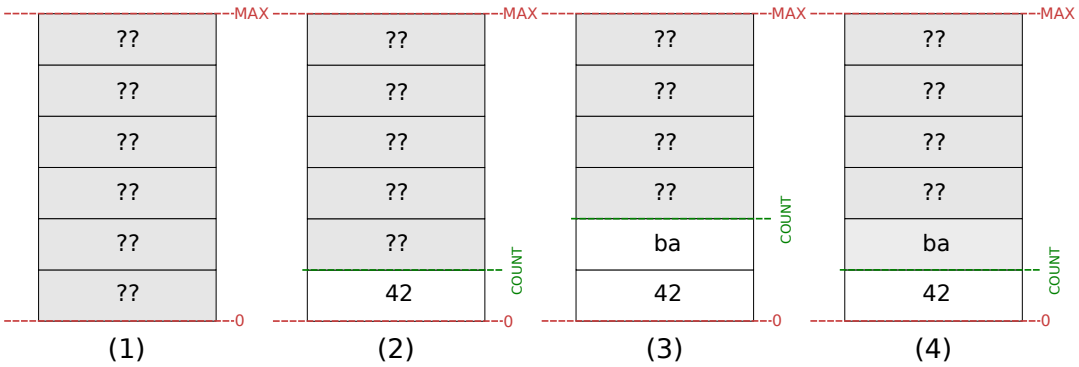

FIGURE 4.5 – Différents états d'une pile mémoire

Lors de la première ligne, notre pile est vide et les valeurs contenues sont quelconques. En empilant une valeur (42), nous connaissons le contenu du premier emplacement. Si nous continuons à empiler des valeurs, les valeurs déjà existantes doivent rester identiques. Le dépilement ne supprime pas les données mais simplement décrémente le nombre d'éléments. À partir de cet exemple, nous pouvons déduire les propriétés dont nous avons besoin pour exprimer les différents états : pile vide, pile pleine et intégrité des données.

Lorsque la pile est vide, elle ne contient aucun élément donc son nombre d'éléments est égal à zéro : c'est l'objectif du prédicat `is_empty_v`. Lorsqu'elle est pleine, son nombre d'éléments est au moins égal à son nombre maximal d'éléments comme le spécifie le prédicat `is_full_v`. Enfin, lors d'un empilement ou d'un dépilement seul le sommet de la pile peut être modifié : il s'agit du prédicat `buffer_lr_not_changed_v` où `n` correspond au sommet.

```
let is_empty_v (#a:eqtype) (h:mem) (s:(stack a)) : GTot Type0 =
    U32.v s.count = 0

let is_full_v (#a:eqtype) (h:mem) (s:(stack a)) : GTot Type0 =
    U32.v s.max ≤ U32.v s.count

let buffer_lr_not_changed_v (#a:eqtype) (h0 h1:mem) (b:buffer a)
    (n:nat{n < length b})
    : GTot Type0
=
    (∀ i. i < n ⟹
        (get h0 b i) = (get h1 b i))
```

Il est important de noter que ces prédicats traduisent grâce à `U32.v` les nombres machine – c'est-à-dire les nombres tels qu'encodés dans un processeur – en nombres naturels ($\mathbb{N}$).

Le prédicat `buffer_lr_not_changed_v` nous permet d'aborder deux subtilités : 1) seules les valeurs en dessous du sommet restent inchangées, 2) la comparaison se fait sur deux instances de la mémoire. Dans le premier point, les données qui sont au-dessus n'ont pas de sens et donc ne sont pas importantes. Pour le second point, le modèle de mémoire utilisé permet de tenir compte

de toutes les modifications sur la mémoire d'un programme. Cela nous permet de comparer les données de la pile dans la mémoire avant et après l'appel d'une fonction. Si ces données sont identiques, nous dirons qu'elles sont stables.

Maintenant que nous pouvons aussi raisonner sur les états possibles d'une pile mémoire, nous allons décrire comment la modifier. Nous allons décrire les fonctions d'empilement `push` et de dépilement `pop`. Leur signature sans les pré-/post-conditions peut s'écrire en F⋆ comme suit :

```
val push : (#a:eqtype) → (s:pointer (stack a)) → (v:a) → ST (unit)
val pop :  (#a:eqtype) → (s:pointer (stack a)) → ST (a)
```

Dans une première étape, nous considérerons l'empilement qui se décompose en trois lignes de code : le déréférencement de la structure, l'ajout en mémoire de la valeur concernée (`v`) puis l'incrémentation du nombre d'éléments. Dans le code suivant, nous montrons également comment mettre à jour l'élément dans le tableau `data` et comment mettre à jour le champ `count`. Nous faisons remarquer que la syntaxe en F⋆ est différente pour ces deux dernières opérations – l'une écrit en mémoire et l'autre met à jour une structure.

```
let s0 = !*s in
s0.data.(s0.count) ← v;
s *= {s0 with count=(U32.add s0.count 1ul)}
```

Si le code présenté ci-dessus semble correct, il est en réalité dangereux. En effet, aucun contrôle n'est fait pour vérifier la validité de la pile. Cependant comme nous l'avons vu avant, il n'est pas possible de construire une pile invalide, mais pour vérifier la cohérence du programme, F⋆ a besoin d'une modélisation de toutes les étapes. Cette modélisation peut se faire en rajoutant un code qui va forcer les tests et donc implicitement raffiner les données – enrichir les données avec la connaissance induite par le test – pour qu'elles puissent être validées. Cependant, cette approche est coûteuse en instructions et elle nécessite de prendre en compte le cas où le programme est en échec. Par exemple, si nous avons une pile mémoire qui a une capacité de deux éléments, le programme suivant avec les tests aurait un surcoût pour les deux premiers empilements, mais il empêcherait le problème dû à l'empilement d'un troisième élément. Cependant, sans ces tests ce programme est dans un état incorrect au troisième empilement.

```
; initialization     count = 0   max = 2
push 1  ;            count = 1   max = 2
push 2  ;            count = 2   max = 2
push 3  ;            count = 3   max = 2 (Incorrect or exception)
```

Afin de ne vérifier que les cas qui nous intéressent, nous allons plutôt modéliser le comportement du code. Ceci permet de repousser le contrôle des données (dans certains cas, de le supprimer complètement) et de pouvoir indiquer à F⋆ comment évoluent nos données. Dans ce cas, nous allons ajouter une précondition qui indique clairement que notre fonction d'empilement `push` ne peut être appelée que sur des piles bien formées et qui ne sont pas pleines.

```
(requires λ h →
    wellformed h s
    ∧ ¬(is_full_v h (deref h s))
)
```

Maintenant nous pouvons vérifier que notre fonction `push` n'est appelée que dans les cas où la pile est correcte. Dans ce cas F⋆ ne prendra en compte que les conditions de l'appel, mais il ne pourra toujours pas vérifier une séquence d'empilements successifs puisqu'il ne sait pas quel impact cette fonction a sur notre pile. En fait, il nous faut lier la situation initiale de `push` avec sa situation finale. Nous modélisons une postcondition qui indique à F⋆ l'impact exact de notre fonction :

```
(ensures λ h0 x h1 →
    wellformed h1 s
    ∧ (let s0 = (deref h0 s) in
        let s1 = (deref h1 s) in
        U32.v s0.count + 1 = U32.v s1.count
        ∧ U32.v s0.max = U32.v s1.max
        ∧ U32.v s1.max = length s1.data
        ∧ get h1 s1.data (U32.v s0.count) = v
        ∧ modifies (loc_union (loc_buffer s) (loc_buffer s0.data)) h0 h1
        ∧ buffer_lr_not_changed_v #a h0 h1 s0.data (U32.v s0.count)
    )
)
```

Ce code va indiquer que dans l'état final de `push` la pile mémoire est bien formée, puis il compare la pile entre la version d'avant l'appel à `push` (`s0`) et la version d'après l'appel à `push` (`s1`). Nous indiquons également que le nombre d'éléments est incrémenté et quelle valeur est mise à jour dans le nouvel emplacement. Nous précisons aussi que certaines données restent constantes, comme le nombre maximal d'éléments, la taille du tableau ou les données stockées précédemment (via `buffer_lr_not_changed_v`). De plus, le prédicat de L⋆ `modifies` indique quelles parties de la mémoire de notre programme sont modifiées.

Ces modifications permettent à F⋆ de rejeter notre troisième empilement. Cela nous pousse à modifier notre programme d'exemple en tenant compte de la taille de la pile. Il est possible de supprimer l'appel incorrect car dans notre exemple la pile ne pourra jamais empiler de troisième élément. Il n'est pas possible pour F⋆ de valider un programme qui fait trois empilements, car l'appel à `push 3` casse les contraintes sur la taille de la pile. Nous donnons ci-dessous la version de notre programme d'exemple corrigée :

```
; initialization     count = 0   max = 2
push 1   ;           count = 1   max = 2
push 2   ;           count = 2   max = 2
;push 3
```

De façon similaire, nous pouvons définir une fonction `pop` pour dépiler les valeurs de la pile. Le code est l'opposé de celui concernant l'empilement : le déréférencement, la décrémentation du nombre d'éléments et le retour de la valeur stockée en sommet de pile.

```
let s0 = !*s in
let c = s0.count `U32.sub` 1ul in
s *= {s0 with count=c};
s0.data.(c)
```

Comme pour l'empilement, ce code est correct mais pas dans tous les états de la pile. Nous devons nous assurer que notre pile possède au moins un élément et qu'elle est bien formée.

```
(requires λ h →
    wellformed h s
    ∧ ¬(is_empty_v h (deref h s))
)
```

Pour l'impact de la fonction `pop`, nous indiquons les changements entre avant et après l'appel à la fonction. Nous faisons remarquer que le prédicat `modifies` ne désigne que le pointeur sur notre structure. Cela est normal puisque seul le champ contenant le nombre d'éléments est modifié.

```
(ensures λ h0 x h1 →
    wellformed h1 s
    ∧ (let s0 = (deref h0 s) in
        let s1 = (deref h1 s) in
        U32.v s0.count - 1 = U32.v s1.count
        ∧ U32.v s0.max = U32.v s1.max
        ∧ U32.v s1.max = length s1.data
        ∧ get h0 s0.data (U32.v s1.count) = x
        ∧ buffer_lr_not_changed_v #a h0 h1 s0.data (U32.v s1.count)
        ∧ modifies (loc_buffer s) h0 h1
    )
)
```

Nous faisons observer que contrairement à la postcondition de `push` où nous vérifions la stabilité sur `s0.count`, nous la vérifions sur `s1.count`. Sans cette modification, F⋆ n'aurait pas validé le code, car nous ne respectons pas le type de `buffer_lr_not_changed_v`. Cependant, la raison logique pour laquelle nous travaillons sur `s1.count` est simple : l'élément dépilé ne fait plus partie de la pile.

4.4.3 Exemple de lemme

Notre pile est maintenant vérifiée pour l'empilement et le dépilement. Aussi, nous aimerions vérifier que pour toute pile, l'ajout d'un élément puis sa suppression nous laisse une pile équiva-

lente à celle que nous avions au départ. Nous pouvons écrire le programme suivant qui exprime cette propriété :

```
let push_pop_equiv (#a:eqtype) (v:a) (s:pointer (stack a)) : ST (unit) =
    push s v;
    let x = pop s in
    assert(x == v) (* x and v are equal *)
```

En fait, ce programme en l'état est incorrect. En ajoutant la précondition et la postcondition, nous pouvons raisonner sur l'impact de ce programme et vérifier si notre propriété est valide. Comme notre programme empile un élément, nous devons exclure toutes les piles pleines (grâce à `¬(is_full_v h (deref h s))`). Nous vérifions aussi que la pile avant et après notre programme possède le même nombre d'éléments (`U32.v s0.count = U32.v s1.count`) et que les éléments correspondants sont identiques (en utilisant `buffer_lr_not_changed_v`).

```
(requires λ h →
    wellformed h s
    ∧ ¬(is_full_v h (deref h s))
)
(ensures λ h0 x h1 →
    wellformed h1 s
    ∧ (let s0 = (deref h0 s) in
        let s1 = (deref h1 s) in
        U32.v s0.count = U32.v s1.count
        ∧ U32.v s0.max = U32.v s1.max
        ∧ U32.v s1.max = length s1.data
        ∧ buffer_lr_not_changed_v #a h0 h1 s0.data (U32.v s1.count)
        ∧ modifies (loc_union (loc_buffer s) (loc_buffer s0.data)) h0 h1
    )
)
```

4.4.4 Extraction, optimisation et intégration

Le but premier de la vérification est de pouvoir exprimer un programme ayant des propriétés logiques mais une des raisons pour lesquelles F⋆ a été choisi c'est qu'il est possible d'extraire un programme en langage C. Ce n'est pas juste une compilation du F⋆ vers le C, mais une traduction. Il s'agit de traduire la syntaxe – les mots-clefs sont remplacés – pour l'adapter au langage cible. De fait, le code généré est très similaire au code source. De plus, nous ne traduisons pas en langage machine mais en un autre langage : il s'agit toujours d'un code lisible par l'être humain – le code n'est toujours pas traduit dans un langage machine. L'implémentation de `push` est le résultat de cette traduction.

```
void push__uint32_t(stack__uint32_t *s, uint32_t v1)
{
    stack__uint32_t s0 = *s;
    s0.data[s0.count] = v1;
    stack__uint32_t uu____0 = s0;
    *s
    =
        (
            (stack__uint32_t){
            .max = uu____0.max,
            .count = s0.count + (uint32_t)1U,
            .data = uu____0.data
            }
        );
}
```

Si nous comparons le code ci-dessus avec une fonction similaire écrite en C présentée ci-dessous, une personne expérimentée en C remarquera que la différence est en réalité très légère. À part une variable en trop (`uu____0`), le code est équivalent et le compilateur pourra optimiser cette partie afin d'avoir le moins de surcoût possible.

```
void push(stack_t *s, int32_t v){
    s->data[s->count++] = v;
}
```

Comme nous l'avons énoncé précédemment, contrôler chaque empilement représente un coût non négligeable. Comme exemple nous imaginons une fonction qui additionne les éléments restant dans une pile. Nous supposons que l'addition s'effectue sur deux ou trois éléments seulement.

Normalement dans l'industrie les spécifications sont faites pour indiquer au développeur ce que la fonction attend et produit. Dans notre exemple nous allons supposer que l'implémentation par l'utilisateur est totalement naïve. L'objectif est d'additionner les trois (ou deux) éléments restants. Comme notre fonction `pop` ne fait aucun test, nous testons tous les cas possibles.

```
let example_add (s:pointer (stack U32.t)) : ST (U32.t) =
    let a = if not (is_empty s) then pop s else 0ul in
    let b = if not (is_empty s) then pop s else 0ul in
    let c = if not (is_empty s) then pop s else 0ul in
    U32.add_underspec a
        (U32.add_underspec b c)
```

Cependant, nous savons par la spécification de notre exemple que nous avons une pile qui a au moins deux éléments et trois au plus. Nous écrivons alors la précondition suivante :

```
(requires λ h →
    wellformed h s
    ∧ (U32.v (deref h s).count = 2 ∨ U32.v (deref h s).count = 3)
)
```

De même nous spécifions la postcondition de manière à refléter un dépilement de tous les éléments. Il est intéressant de remarquer que la stabilité des données n'a plus vraiment de sens car la pile ne contient plus aucune donnée puisqu'elle sera totalement dépilée.

```
(ensures λ h0 x h1 →
    wellformed h1 s
    ∧ (let s0 = (deref h0 s) in
        let s1 = (deref h1 s) in
        (U32.v s1.count = 0)

        ∧ U32.v s0.max = U32.v s1.max
        ∧ U32.v s1.max = length s1.data
        ∧ U32.v s1.max = length s0.data

        ∧ (modifies (loc_buffer s) h0 h1)
    )
)
```

Maintenant nous pouvons un à un retirer les tests et voir jusqu'où la vérification est toujours valide. Nous pouvons alors découvrir que seul le troisième test est nécessaire. En effet, si le développeur retire tous les tests, F⋆ échouera pour la valeur de `c` puisque la pile peut être vide dans certains cas.

```
let example_add (s:pointer (stack U32.t)) : ST (U32.t) =
    let a = pop s in
    let b = pop s in
    let c = if not (is_empty s) then pop s else 0ul in
    U32.add_underspec a
        (U32.add_underspec b c)
```

Cet exemple montre à quel point la modélisation peut préciser le raisonnement formel. Bien sûr cet exemple reste un cas particulier et dans de nombreuses situations, l'absence d'information sur les données possibles nous oblige à avoir des tests à certains endroits. Maintenant que nous avons montré comment modéliser un programme et un état logiciel ou matériel, nous pouvons expliquer comment utiliser ces deux formes de modélisation pour modéliser un système *cyberphysique* tel que le moteur pas à pas.

Deuxième partie

Vérification et systèmes cyberphysiques

Dans cette partie, nous présentons un premier aspect de nos travaux : la vérification formelle de programmes embarqués. En effet, la vérification de systèmes embarqués est essentielle, et F⋆ présente une approche compatible avec les contraintes de la machine grâce à KreMLin [1]. Cette partie présente trois contributions :

— Une modélisation de la carte Arduino. Cette contribution montre comment relier la vérification formelle en F⋆ aux contraintes matérielles de l'architecture utilisée. Il s'agit alors d'illustrer une implémentation en F⋆ d'une modélisation matérielle. Il y a deux objectifs derrière cette contribution : l'un est de formaliser l'état de l'Arduino à chaque ligne d'un programme et l'autre de refuser automatiquement un programme qui ne respecte pas les contraintes de la carte. Cette contribution questionne sur la portée de la vérification : doit-elle s'arrêter à la frontière entre la machine et l'environnement ?

— Une modélisation d'un moteur pas à pas. Cette contribution illustre la manière dont F⋆ peut vérifier des propriétés *cyberphysiques* dans les programmes suivant l'évolution de l'environnement physique. Il s'agit de la contribution *cyberphysique* de ces travaux et qui est présentée dans l'article (Talpin et al. 2019) et qui répond à la question soulevée par la première contribution.

— La vérification d'un gestionnaire mémoire système. Bien que moins axée sur l'approche *cyberphysique*, cette contribution reprend un résultat de l'état de l'art (Choudhuri et Givargis 2005) proposant une version formellement vérifiée. Contrairement aux précédentes contributions, il s'agit de modéliser un composant et non l'interaction avec l'environnement. Ce résultat n'a pas été publié.

Cette partie se décompose en deux chapitres. Le premier – chapitre 5 – concerne l'explication de la modélisation d'un logiciel embarqué. Ce chapitre traite de la portée de la formalisation : il est possible de modéliser des mécaniques du monde physique afin de prévoir l'évolution possible des valeurs.

Alors que l'objectif du chapitre 5 est de montrer comment la modélisation est définie pour une architecture, le chapitre 6 illustre l'utilisation de la modélisation pour des applications concrètes. Ce chapitre présente deux applications précises : un moteur pas à pas en section 6.1 et un gestionnaire de mémoire en section 6.2.

Pour synthétiser, la partie II traite de l'utilisation des méthodes formelles dans le cadre de l'informatique embarquée. Elle présente la vérification d'une architecture, d'un composant *cyberphysique* et d'un composant système.

1. Le compilateur KreMLin a été renommé en KaRaMeL le 21 mars 2022.

Vérification de l'embarqué

Comme nous l'avons vu précédemment, F⋆ est un langage qui permet de vérifier des programmes – chapitres 2 et 3 – et d'extraire leur code de manière très synthétique – section 4.3 du chapitre 4. Dans ce chapitre, nous présentons la manière de vérifier une carte comme l'Arduino en F⋆. Cette partie met également en lumière comment relier la vérification à une extraction de code en tenant compte du matériel utilisé. Cette partie soulève un aspect des systèmes *cyber-physiques* : comment vérifier un programme qui manipule des entrées/sorties dépendant d'un environnement physique.

5.1 Connexion entre la vérification et le matériel

L'Arduino est un microcontrôleur 8 bits basé sur les processeurs Atmel. Nous utilisons l'ATmega328p. Ce processeur – conçu dans les années 1990 (*Chip Hall of Fame : Atmel ATmega8* 2017) – est extrêmement basique : pas de gestionnaire mémoire, pas de confinement entre tâches et très peu de ressources – 2 Ko de SRAM et 32 Ko de mémoire Flash (*List of Arduino boards and compatible systems* 2020). Si d'un point de vue sécuritaire ce processeur n'est pas intéressant, nous croyons que la vérification formelle peut augmenter de tels systèmes. Cela permettrait d'utiliser des programmes sûrs dans des applications moins coûteuses et plus miniaturisées. En effet, les fonctionnalités de sécurité ajoutent une complexité qui a un coût financier et énergétique.

5.1.1 Entrées/Sorties sur microcontrôleur

La programmation embarquée nécessite, en général, d'interagir avec des capteurs et des actionneurs. C'est un domaine où les entrées/sorties peuvent être très variées. Nous les appelons "pattes" en référence à leur forme. Dans le cas d'un microcontrôleur comme l'Arduino, il existe trois types registres – variable temporaire définie matériellement – qui gèrent les états des pattes. Ils sont notés *B*, *C* et *D*. Les registres *B* et *D* manipulent les pattes numériques qui n'ont que deux états possibles *HIGH* et *LOW* (*Port Registers* 2020). Le registre *C* gère les pattes analogiques qui retournent un entier avec une précision sur 10 bits par défaut, mais peut être étendu sur certains modèles (*Arduino Documentation - analog read* 2020).

Il est possible de lire ces registres en langage C via des macros définies dans le code dans la bibliothèque AVR. Il existe trois types d'accès : *PORT* qui permet d'écrire, *PIN* qui permet de lire et *DDR* qui détermine si la patte est dans un mode de lecture ou d'écriture. La figure 5.1 donne une vue de tous les registres. Il y a donc 9 registres en tout. Nous précisons que les

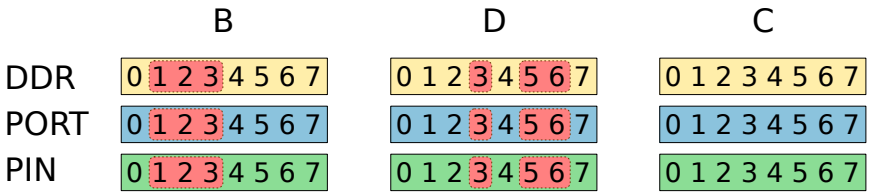

FIGURE 5.1 – Tous les registres possibles

registres sont tous d'une taille de 8 bits, nous avons un bit par patte. Lorsqu'un programme lit l'état d'une patte numérique, il lira simultanément l'intégralité des autres pattes sur le même registre. Cependant, les bibliothèques cachent ce détail au développeur.

L'utilisateur d'Arduino soulignera que certaines pattes peuvent être utilisées de deux manières possibles. Par exemple, les bits marqués en rouge sur 5.1 sont capables de gérer matériellement le Pulse Width Modulation (PWM) en plus d'être des pattes numériques.

Notons qu'en regardant la disposition des pattes sur une Arduino (figure 5.2), on voit que les registres contiennent plus de bits que nécessaire. Certains bits peuvent être inutilisés ou réservés pour un usage interne au microcontrôleur, qui rappelons-le n'est pas open source. Dans le code suivant, nous donnons une implémentation de la fonction qui lit la valeur d'une patte numérique.

```
unsigned char digitalRead(unsigned char pin){
    if (pin < 8)
        return ((PIND >> pin) & 0x01);
    if (pin < 16)
        return ((PINB >> (pin - 8)) & 0x01);
    return LOW;
}
```

Bien qu'il soit possible de modéliser du code spécifique à une architecture spécifique avec Vale (FROMHERZ et al. 2019), nous utilisons l'architecture Atmel qui n'est pas supportée pour l'instant. Nous allons donc présupposer – c'est-à-dire considérer comme correctes – toutes les fonctions d'entrées/sorties déjà existantes en C comme `digitalRead` précédemment décrite. Cependant, nous allons montrer comment modéliser par un code leur comportement pour exhiber leur impact. Il est important de comprendre qu'une fonction peut modifier l'état global d'un système. Nous modélisons cet impact sur le système avant de vérifier la cohérence du programme. Nous présentons deux approches : l'une utilisée depuis le début dans la communauté de F⋆, l'autre qui utilise une nouvelle manière de déclarer les effets dans F⋆, appelée DM4F.

5.1.2 Approche traditionnelle

Comme expliqué dans le chapitre 2, les effets indiquent comment propager le calcul de la condition la plus faible tout en tenant compte d'un contexte. Initialement les effets comme l'effet

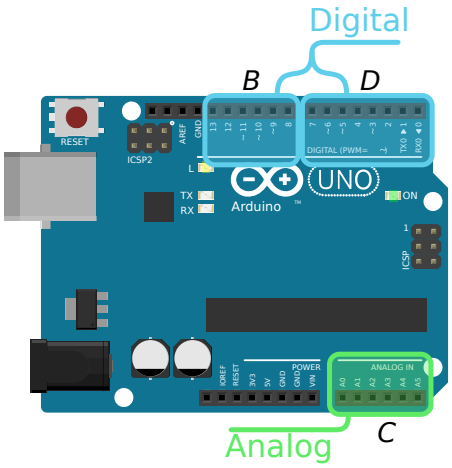

FIGURE 5.2 – Disposition des pattes et registres

STATE ne sont pas réifiables – ils ne sont pas interprétables par F⋆ – ce qui implique de formaliser la spécification des entrées/sorties sans modéliser le comportement. Ainsi, les fonctions contiennent juste une signature sans code associé. Nous illustrons dans le code suivant la fonction qui lit une patte numérique sur l'Arduino. Cette fonction est plus typée que la version C, ce qui n'est vrai qu'en F⋆, car après la vérification le programme utilise le typage du langage C.

```
assume val digitalRead : (p:digital_pin) → Arduino (digital_value)
    (requires λ b →
        board_initialized b
        ∧ getPinMode p b = INPUT)
    (ensures λ b0 x b1 →
        getPinValue p b1 = x)
        ∧ modifies b0 b1 [])
```

Dans cette situation, F⋆ va simplement raisonner sur la pré- et la post-condition. Cela permet de considérer le code comme correct et de l'utiliser tel qu'il est censé fonctionner. C'est pratique dans notre cas, puisque le code existant n'est pas exprimable en F⋆ pour le moment. Cependant, F⋆ ne sera pas capable de comprendre ce qui se passe à l'intérieur de ces fonctions. En fait, F⋆ perd toute information sur le contexte que la spécification ne précise pas, ce qui est la principale limite de cette approche.

5.1.3 Nouvelle approche

Contrairement à l'approche précédente, l'impact d'une fonction ne sera pas uniquement spécifié, mais aussi exprimé en F⋆. Cela permet de vérifier la cohérence de la spécification et de conserver les informations du contexte. Cette approche est basée sur une nouvelle définition des effets dans F⋆, appelée DM4F (MAILLARD et al. 2019). Le code suivant montre la même fonction que dans le paragraphe précédent.

```
let digitalRead (p:digital_pin) : Arduino (digital_value)
    (requires λ b →
        board_initialized b
        ∧ getPinMode p b = INPUT)
    (ensures λ b0 x b1 →
        getPinValue p b1 = x)
        ∧ modifies b0 b1 [])
=
    let i,b = STATE?.get () in                    (* ouverture du contexte *)
    let Some (Digital pin m v) = get (i,b) p in (* valeur simulée *)
    v
```

Nous faisons remarquer qu'en plus de la spécification, le code lit la valeur stockée dans la modélisation de l'Arduino grâce à la fonction `STATE?.get`. De cette manière, il est possible de conserver les informations du contexte tout en pouvant tracer l'origine de la valeur contenue dans `v`.

L'intérêt de cette approche, qui modélise précisément l'impact des entrées/sorties, n'est pas d'aider uniquement F⋆ dans l'évaluation de l'état du programme : nous pouvons également ajouter à la modélisation des valeurs qui peuvent dépendre d'une simulation physique et donc modéliser des processus *cyberphysiques* tels que des processus biologiques ou des fautes matérielles.

Cependant, comme le code existant n'est pas exprimable en F⋆, nous devons supprimer cette fonction à la compilation sans enlever les appels vers elle. Ainsi, le programme continuera d'appeler la fonction `digitalRead` qui sera une fonction propre à l'architecture.

5.2 Modélisation d'une Arduino

Comme expliqué précédemment, la qualité de vérification d'un programme F⋆ dépend de sa modélisation et de celle de son environnement. Dans cette section, nous présentons une modélisation de l'Arduino en trois axes : la modélisation des pattes, la modélisation de l'état et la gestion des entrées/sorties.

Nous pouvons nous questionner sur la valeur ajoutée par la modélisation du microcontrôleur. En effet, si un programme est valide, nous nous attendons à ce qu'il n'échoue jamais. En réalité, la notion de justesse dépend grandement de l'environnement. Nous illustrons cela avec les accidents des Boeing 737 Max en 2019. Ce modèle d'avion est équipé du Maneuvering Characteristics Augmentation System (MCAS) qui est un système d'évitement d'un décrochage de l'avion – chute de l'avion suite à une vitesse insuffisante et un angle d'attaque trop élevé (Johnston et Harris 2019).

Le système MCAS ne lit qu'une des sondes permettant de mesurer le décrochage. Or si cette sonde est défectueuse – ce qui peut arriver – le système est incapable de s'en rendre compte. Cela

s'est produit dans les deux accidents fatals qui ont conduit Boeing à rappeler tous les modèles 737 Max. Cependant, la modélisation du microcontrôleur permet d'ajouter des incohérences liées au monde physique comme les fautes matérielles. Cela pourrait, par exempe, forcer les développeurs à multiplier les sources d'informations afin d'obtenir une valeur confortée par la redondance.

En parallèle de ce problème critique, la certification du 737 Max avait été allégée et la majorité des pilotes n'étaient même pas au courant de l'existence du MCAS. Cela explique pourquoi ce système ne fut jamais désactivé pendant les accidents. Notons que les éléments externes comme la formation des pilotes sont hors de notre sujet d'étude. La faute matérielle introduit un non-déterminisme et nous indiquons en fin de section où il est possible d'ajouter un tel mécanisme dans la vérification afin de prendre en compte ces problèmes.

> Notre travail consiste à modéliser l'état des entrées/sorties dans un microcontrôleur de type Arduino. Grâce à une telle modélisation, il est possible de vérifier la bonne utilisation des entrées/sorties mais aussi comment elles peuvent se comporter dans un cadre d'interaction *cyberphysique*, ce qui est novateur. Cette section présente un travail qui a été publié dans (Talpin et al. 2019).

5.2.1 Définition des pattes

Dans cette sous-section, nous abordons la sécurité de la manipulation du matériel sous l'aspect syntaxique. Dans l'architecture Arduino, les pattes sont identifiées par un caractère non signé dont la valeur dépend de la configuration matérielle. De plus, les fonctions d'entrées/sorties prennent n'importe quel identifiant, alors que le microcontrôleur en a un nombre spécifique qui peut changer d'un modèle à l'autre. Ajoutons également que nous ne tenons pas compte du fait que les applications embarquées ont un sens spécifique à chaque patte. Ce sens dépend du matériel qui est relié.

Pour illustrer un possible dysfonctionnement, nous présentons le code suivant en C. Ce dernier est parfaitement valide, mais matériellement incorrect tout en étant compilable. En effet, il lit la valeur d'une patte (`0xFF`) qui n'existe pas sur la plupart des modèles. Cependant, pour pallier de telles incohérences, les bibliothèques retournent généralement une valeur par défaut qui est `LOW`.

```
int r = digitalRead(0xFF);
```

Nous montrons comment enrichir le système de types pour encapsuler ces contraintes dans le typage. Dans notre cas, le processeur Atmel328p définit les valeurs suivantes : 0-7 pour les identifiants de type *D*, 8-15 pour les identifiants de type *B* et 160-165 (`0xA0`-`0xA5`) pour les identifiants de type *C*. Nous indiquons alors à F$\star$ quels sont les domaines possibles pour les identifiants des pattes numériques et analogiques. Nous renseignons à F$\star$ les domaines de valeurs que ces types de pattes peuvent gérer.

```
type digital_pin   = p:U8.t{U8.v p < 16}
type digital_value = | LOW | HIGH

type pwm_pin = p:digital_pin{(let x = U8.v p in
    x = 5 ∨ x = 6 ∨ x = 3
    ∨ x = 9 ∨ x = 10 ∨ x = 11)}
type pwm_value = U8.t

type analog_pin   = p:U8.t{U8.v p ≥ 160 ∧ U8.v p ≤ 165}
type analog_value = U16.t

type pin_mode = | INPUT | OUTPUT
```

Dans le code précédent, le type `U8.t` correspond au type `unsigned char` en C qui est le type pour les caractères non signés sur 8 bits. De même, le type `U16.t` est un nombre 16 bits non signé, ce qui correspond aux valeurs analogiques encodées sur 10 bits. Concernant le type `digital_value`, il est intéressant de noter que ce n'est pas par hasard que `LOW` est défini avant `HIGH` : ce type sera traduit par une énumération en C. La valeur du premier élément vaudra 0 et celle du second vaudra 1, ce qui correspond aux valeurs attendues par les pattes numériques. Il en est de même pour le type `pin_mode`.

Nous définissons également les pattes numériques compatibles avec le PWM. Le domaine de valeur pour ces pattes est un pourcentage où 0 % et 100 % correspondent à 0 et 255 respectivement. Ce système permet d'envoyer des impulsions sur la patte afin de coder une proportion. Cela permet d'envoyer 256 informations différentes sur 1 bit de donnée.

Cependant, comment allons-nous déterminer si un caractère quelconque correspond à une patte numérique, PWM ou analogique ? Il nous suffit de définir une fonction pour chaque type, qui retourne le fait que son paramètre correspond ou non à la description. Nous donnons le code ci-dessous pour de telles fonctions. Notons que la définition de `isPWM` est différente de la définition du type `pwm_pin`. Cette différence s'explique par un code plus simple à exprimer qui est strictement équivalent. Nous montrerons cette équivalence.

```
let isDigital (x:U8.t) : bool = x `U8.lt`  16uy
let isPWM (x:U8.t) : bool =
    x = 5uy || x = 6uy || x = 3uy
    || (x `U8.gte` 9uy && x `U8.lte` 11uy)
let isAnalog  (x:U8.t) : bool = x `U8.gte` 160uy &&  x `U8.lte` 165uy
```

Comment pouvons-nous être sûrs de la justesse de ces fonctions ? En effet, le typage du retour n'indique absolument rien. Raffiner le retour permettrait de vérifier la fonction mais alourdirait la vérification en précisant trop les types de retour. Pour y remédier nous utilisons une preuve automatique extrinsèque décrite ci-dessous. Par convenance, nous n'avons pas nommé les lemmes.

```
let _ =
    assert( ∀ (x:U8.t).
        isDigital x ⟺ (∃ (y:digital_pin). U8.v x = U8.v y));
    assert( ∀ (x:U8.t).
        isPWM x ⟺ (∃ (y:pwm_pin). U8.v x = U8.v y));
    assert( ∀ (x:U8.t).
        isAnalog x ⟺ (∃ (y:analog_pin). U8.v x = U8.v y));
    ()
```

Chaque assertion demande à F⋆ de prouver son contenu. Prenons l'exemple des pattes numériques, nous disons basiquement que pour toute valeur, si la fonction `isDigital` retourne vrai pour une valeur quelconque alors il doit exister la même valeur dans le domaine de valeurs du type `digital_pin`, sinon il ne doit pas exister de valeur dans ce domaine. Grâce à ces assertions nous pouvons être sûrs de la justesse du code.

Nous pourrions déjà typer nos fonctions d'entrées/sorties, ce qui est un progrès par rapport au typage actuel du langage C mais nous présentons dans le paragraphe suivant la modélisation de l'état du microcontrôleur.

5.2.2 Modélisation du microcontrôleur

Nous avons répondu à la question syntaxique précédemment, nous répondons à la question sémantique dans cette sous-section. En effet, beaucoup de personnes sont satisfaites avec le typage du langage C qui donne une grande liberté. Dans F⋆ nous pouvons aller plus loin et montrer la cohérence en termes d'utilisation des fonctions. Sur l'Arduino, il est possible via la fonction `pinMode` de changer le mode d'accès d'une patte. Considérons le code C suivant :

```
pinMode(5, OUTPUT);
int r = digitalRead(5);
```

Dans le code ci-dessus, nous configurons une patte en écriture puis nous lisons sa valeur. Ce code est syntaxiquement correct mais sémantiquement incorrect : les deux actions sont contradictoires. Nous présentons dans cette partie comment modéliser l'état de l'Arduino pour prendre en compte cette cohérence. Pour cela nous donnons une représentation abstraite des pattes :

```
type pinout =
    | Digital : digital_pin → pin_mode → digital_value → pinout
    | PWM     : pwm_pin     → pin_mode → pwm_value     → pinout
    | Analog  : analog_pin  → pin_mode → analog_value  → pinout

type board =  bool * (U8.t → option pinout)
```

Le code précédent définit les pattes abstraites grâce au type `pinout`. Chaque type de patte est typé par rapport à la spécification du microcontrôleur ainsi il n'est pas possible de modéliser des

pattes incorrectes. Pour chaque type de patte, nous définissons également les valeurs ainsi que le mode dans lequel la patte se trouve. Le type `board` modélise le microcontrôleur avec un booléen qui renseigne si l'initialisation a bien eu lieu et une fonction qui prend un numéro de patte et retourne peut-être l'état d'une patte abstraite. Cette fonction retourne l'état actuel de la patte s'il existe. Si personne n'a configuré une certaine patte nous souhaitons n'avoir aucune représentation abstraite de celle-ci. Cette représentation est appelée réflexion puisqu'elle agit comme un miroir de l'état de l'Arduino dans les préconditions les plus faibles. Ainsi, la modélisation d'une Arduino non initialisée sera exprimée ainsi :

```
let initial_board : board = false, (λ _ → None)
```

Lorsque nous voulons savoir si l'Arduino n'est pas initialisée nous pourrions simplement déterminer si la carte actuelle est identique à la carte initiale. Cependant, nous ne regardons que l'état d'initialisation :

```
let board_initialized b =
    let initialized, _ = b in
    initialized = true
```

Avant d'aller plus loin, nous détaillons un prédicat appelé `modifies` que nous utilisons beaucoup. Ce prédicat prend deux modélisations d'Arduino et une liste de pattes. Il vérifie que les différences entre ces deux modélisations sont potentiellement uniquement les pattes indiquées par la liste de numéros.

```
let modifies (b0 b1:board) (l:list u8) =
    (fst b0 == fst b1)
    ∧ (∀ (k:u8{¬(L.contains k l)}).
        (snd b0) k == (snd b1) k)
```

Pour ajouter un élément, nous utilisons l'aspect fonctionnel de F⋆ en encapsulant la mémoire dans une nouvelle fonction. Nous pouvons le faire, car ce code n'est pas extrait en C. Il sert uniquement pour vérifier l'implémentation qui sera extraite. Le code suivant prend un microcontrôleur modélisé et retourne la modélisation du même microcontrôleur avec l'ajout d'un nouvel élément. La postcondition indique les différences entre la nouvelle et l'ancienne modélisation. Nous faisons observer l'utilisation de `modifies` dans la post-condition.

```
let set (b:board) (v:pinout) : Pure (board)
    (requires ⊤)
    (ensures λ x →
        (get x (getPin v)) == Some v
        ∧ (fst b) == (fst x)
        ∧ (modifies b x [(getPin v)]))
=
    (fst b), (λ r →
                if r = (getPin v) then
```

```
            Some v
        else
            (snd b) r)
```

Comme la modélisation d'un microcontrôleur utilise un couple de valeurs – l'initialisation et la modélisation des pattes – nous accédons au premier élément par la fonction `fst` et au second élément par la fonction `snd`.

Pour illustrer la façon dont la fonction `set` s'exécute, nous considérons la modélisation des pattes dans le code suivant où seule la patte 5 a été mise à jour (nous l'appelons `b1`). Si nous appelons `b1` avec 5, nous obtenons `Some A`. Toutefois avec n'importe quel autre numéro, `b1` retournera `None`. Pour mettre à jour `b1`, nous définissons `b2` qui contient `b1` mais retourne la nouvelle valeur B de la patte 5. Dans ce cas, appeler `b2` avec 5 retournera `Some B`.

```
let b1 = (λ x → if x = 5uy then Some A else None) in
let b2 = (λ z → if z = 5uy then Some B else (b1 z)) in
assert(b1 5uy == Some A);
assert(b2 5uy == Some B)
```

Nous faisons remarquer que pour récupérer un élément il suffit d'appeler la fonction qui contient toutes les pattes. Ainsi pour récupérer l'état d'une patte, il nous suffit d'appeler le second élément de la modélisation du microcontrôleur. Voici le code qui fait cela :

```
let get (b:board) (p:u8) : option pinout = (snd b) p
```

Maintenant que nous savons mettre à jour et lire l'état d'une patte, nous pouvons modéliser les entrées/sorties de l'Arduino. Pour cela nous décrivons un effet qui permet de modéliser l'impact d'un programme sur le microcontrôleur. Notre effet contient juste l'état actuel du microcontrôleur et nous le construisons grâce à DM4F.

Dans le code suivant, nous montrons comment définir un tel effet. Nous devons au moins exprimer trois éléments : la déclaration de l'effet, l'opération d'élévation – appelée `lift` – et l'expression de la propagation de la condition la plus faible.

La première étape nous permet de donner le nom et le type du contexte. Dans notre cas, le contexte est juste une modélisation de l'Arduino. La fonction `S.STATE_h` est un constructeur DM4F générique qui construit un effet pour nous.

```
total new_effect STATE = S.STATE_h board
```

Dans la deuxième étape, nous élevons le contexte afin qu'il soit plus grand que le contexte `Pure`. Ce contexte appelé `Pure` représente l'ensemble des programmes n'ayant pas d'effet. Ils peuvent avoir une pré- et post-condition, mais ils n'ont aucun effet de bord. `lift_pure_state` est la fonction qui indique comment modifier le contexte – ici on ajoute notre modélisation – et l'opérande `sub_effect` permet à du code dans l'effet `STATE` de contenir du code de l'effet `Pure`. En résumé, nous disons comment inclure un code qui n'a pas d'impact sur les entrées/sorties dans notre effet qui modélise l'Arduino. Notons également l'ajout du contexte `h` dans la condition la

plus faible, ce qui permet de faire le lien entre `wp`, la condition la plus faible pure, et `p`, la condition la plus faible de notre effet.

```
unfold let lift_pure_state (a:Type) (wp:pure_wp a) (h:board) (p:STATE?.post a)
    = wp (λ a → p (a, h))
sub_effect PURE ⇝ STATE = lift_pure_state
```

La troisième étape indique comment traduire une fonction sous forme de triplet de Hoare en plus faible précondition. Ici nous disons que la précondition doit être vraie et que la pré- et la post- condition doivent être vraies pour que l'instruction suivante puisse être évaluée.

```
effect Arduino (a:Type) (pre: STATE?.pre) (post: board → a → board → Type0)
    =
       STATE a (λ n0 p → pre n0
            ∧ (∀ a n1.
                pre n0 ∧ post n0 a n1 ⟹ p (a, n1)))
```

Une fois ces trois étapes écrites, nous utilisons l'effet pour définir des entrées/sorties. Nous commençons par la fonction qui initialise le microcontrôleur. Cette fonction doit être appelée avant toutes les autres, car elle configure les valeurs par défaut du microcontrôleur (*Playing with Arduino* 2020). Comme discuté dans la section 5.1, il existe deux manières d'écrire ces fonctions. La nouvelle approche est plus simple, car elle permet de vérifier la cohérence des pré- et post-conditions.

Dans notre exemple, nous écrivons une fonction qui va mettre à jour l'état d'initialisation de la modélisation de l'Arduino. Nous supposons que l'Arduino n'est pas initialisée puis nous l'initialisons en assurant que l'état des pattes reste identique.

```
noextract
let initChip () : Arduino (unit)
    (requires λ b → ¬(board_initialized b))
    (ensures λ b0 x b1 →
        board_initialized b1
        ∧ snd b0 == snd b1)
=
    let i,b = STATE?.get () in
    STATE?.put (true,b)
```

Cependant, le code ci-dessus est loin d'être utilisable dans l'environnement Arduino. En effet, les fonctions `STATE?.get` et `STATE?.set` sont effacées à l'extraction. De fait, la fonction `initChip` a un corps qui ne sert qu'à la vérification et sera extraite comme une fonction vide. La solution consiste à dire à F⋆ de ne pas extraire `initChip` en utilisant le mot-clef `noextract`. Comme les appels existent toujours, nous pouvons ajouter la vraie fonction `initChip` écrite en C à la compilation. Ainsi, le comportement de `initChip` est modélisé, mais l'utilisation de la version dépendante de l'architecture sera celle qui sera utilisée à la fin.

Nous écrivons maintenant les fonctions d'entrées/sorties. Nous commençons par la fonction `pinMode` qui permet de changer le mode d'accès d'une patte. Cette fonction ne peut être appelée que lorsque l'Arduino est initialisée. Après avoir été appelée, elle initialise la patte `p` avec le mode d'accès `m`. N'oublions que seule la patte `p` est modifiée dans l'Arduino pendant l'exécution de cette fonction.

```
noextract
let pinMode (p:u8) (m:pin_mode) : Arduino (unit)
    (requires λ b → board_initialized b)
    (ensures λ b0 x b1 →
        ((isDigital p ∧ ¬(isPWM p)) ⟺ checkAsDigital p b1 (λ v m' → m = m
    '))
        ∧ (isPWM p ⟺ checkAsPWM p b1 (λ v m' → m = m'))
        ∧ (isAnalog p ⟺ checkAsAnalog p b1 (λ v m' → m = m'))
        ∧ (modifies b0 b1 [p]))
=
    let b = STATE?.get () in
    if isPWM p then (
        let j,r = (set b (PWM p m (getValue ()))) in
        STATE?.put (j,r)
    ) else
    if isDigital p then (
        let j,r = (set b (Digital p m (getValue ()))) in
        STATE?.put (j,r)
    ) else
    if isAnalog p then (
        let j,r = (set b (Analog p m (getValue ()))) in
        STATE?.put (j,r)
    ) else ()
```

Le code ci-dessus permet de soulever deux points intéressants. Tout d'abord le typage des pattes. En effet, certaines pattes numériques peuvent aussi être utilisées en PWM, ce qui demande une initialisation différente. Nous montrons comment cette différence doit être prise en compte. Ensuite, le second point concerne les valeurs que contiennent ces pattes. Bien que cela dépende de ce que nous souhaitons modéliser, nous avons décidé d'avoir des valeurs non prévisibles comme c'est le cas sur une Arduino.

La fonction `getValue` retourne une valeur quelconque. En F⋆, il s'agit d'une valeur sur laquelle nous ne pouvons rien inférer : ses valeurs possibles appartiennent à son domaine de valeur défini par le typage. Nous donnons le code correspond à cette fonction :

```
let getValue #a () : a = admit ()
```

Cette fonction n'a aucun sens sur le plan de la programmation. En effet, le code de cette

fonction est présumé – il est vide mais retourne le type correct – ce qui ne pose aucun problème puisqu'il appartient au domaine de la vérification : cette fonction ne sera pas extraite ; elle ne pourrait pas l'être.

Maintenant nous allons présenter comment décrire la lecture d'une patte numérique afin d'expliquer comment vérifier que la fonction est appelée correctement. C'est dans `getValue` que la modélisation physique peut être ajoutée. En effet, lors de la lecture d'une patte, il est possible que des évènements purement physiques se produisent, ce qui peut engendrer une faute matérielle. Dans notre exemple, nous ne modélisons pas ce genre d'évènement mais c'est une possibilité pour de futurs travaux. Le code suivant donne une implémentation :

```
noextract
let digitalRead (p:digital_pin) : Arduino (digital_value)
    (requires λ b →
        board_initialized b
        ∧ (¬(isPWM p) ⟺ checkAsDigital p b (λ _ m → m = INPUT))
        ∧ (isPWM p ⟺ checkAsPWM p b (λ _ m → m = INPUT)))
    (ensures λ b0 x b1 → modifies b0 b1 [])
=
    let i,b = STATE?.get () in
    let Some r = get (i,b) p in
    match r with
        | Digital pin m v → getValue ()
        | PWM pin m v → getValue ()
```

La fonction `digitalRead` permet de lire une patte numérique. Cette fonction implique que l'Arduino est initialisée et que la patte est dans le bon mode d'accès. Nous précisons que le test doit être vrai pour une patte numérique ou une patte PWM. Puis la fonction retourne la valeur de la patte. Dans ce cas, nous retournons une nouvelle valeur alors que la modélisation de la patte possède déjà une valeur. Cela permet de modéliser le fait qu'une lecture est non déterministe. C'est là qu'il est possible de prendre en compte des éléments de la physique qui seraient intégrés dans le contexte `Arduino`.

Par exemple, imaginons une faute matérielle où lire une patte deux fois d'affilée retourne toujours `LOW`. Il nous est possible de modéliser un tel comportement et par conséquent de savoir lorsqu'un programme lira `LOW` à cause de cette faute.

5.2.3 Limites

Nous avons vu comment une Arduino fonctionne et comment l'utiliser en F⋆. Sous certaines conditions cette vérification n'assure plus rien. Par exemple, une fonction mal modélisée pourrait vérifier des propriétés qui n'existent pas. Si `initChip` initialiserait des pattes contrairement à notre modélisation, alors toute notre vérification serait fausse. Tout d'abord, si le code supposé

correct qui gère les entrées/sorties est erroné, la vérification ne peut pas le détecter bien que cela casse certains lemmes. Cela est une limite de la vérification formelle.

Ensuite, la qualité de la vérification dépend de la précision de la modélisation. Bien qu'il soit possible de modéliser un grand nombre de comportements même physiques, il n'est pas possible de tout modéliser. Nous verrons un exemple de modélisation de moteur pas à pas où ce dernier peut être mis en échec par des conditions physiques que nous ne modélisons pas. De plus, le langage F$\star$ ne peut travailler que sur des nombres entiers (`nat` ou `int`). L'absence des nombres réels ne permet pas de vérifier des propriétés continues comme ce que nous pouvons trouver en biologie ou en physique.

Un système *cyberphysique* étant un système informatique qui interagit avec son environnement. La description d'un tel système prend en compte la manière dont l'environnement est impacté, mais également comment celui-ci évolue soit par des évènements extérieurs, soit par l'action du système *cyberphysique* lui-même. Ainsi, la modélisation de notre Arduino peut être agrémentée de notions concernant l'environnement. Par exemple, un drone peut prendre en compte le lien entre la température générée par les moteurs et la météo pour prévenir la surchauffe. Nous verrons des exemples de programmes qui permettent de montrer à la fois comment modéliser et comment interagir avec le monde physique dans le chapitre suivant.

Exemples cyberphysiques et systèmes

Dans ce chapitre, nous étudions concrètement comment utiliser F⋆ dans l'informatique embarquée. Chaque exemple a pour but d'illustrer une réponse possible à un problème connu, de montrer comment cette réponse est extraite en langage C, et de présenter son comportement dans un cadre *cyberphysique* ou système. Dans un premier temps, nous montrons comment formellement vérifier un moteur pas à pas. Puis dans un second temps, nous vérifions un algorithme de gestion mémoire.

6.1 Moteur pas à pas

Dans le monde industriel, il existe beaucoup de solutions pour transformer de l'électricité en énergie mécanique. Les moteurs sont une des solutions. Certains comme les moteurs sans balais sont appréciés pour leur rapport poids/puissance. Ils sont très utilisés dans la motorisation des drones ou des véhicules électriques. Inversement, pour des applications qui nécessitent une grande précision et une durée de vie plus longue, les moteurs pas à pas conviennent mieux. En effet, ils sont peu coûteux, très précis mais lents. Ce qui nous intéresse c'est un risque de blocage qui peut survenir en fonction de l'utilisation. Ce problème est d'ordre *cyberphysique*, car il intervient dans l'interaction entre le programme informatique et le monde physique.

Ces moteurs sont très répandus dans l'industrie. En fait, il est très improbable que la personne lisant ce document n'en ait jamais utilisé, car des outils de tous les jours en sont équipés, comme les imprimantes. Toutefois ces moteurs peuvent être utilisés au sein de systèmes critiques comme des pompes à insuline. Il existe des alternatives modernes à ces moteurs mais nous nous posons la question de la vérification possible d'un tel système pour permettre à des industriels de pouvoir sécuriser des solutions existantes.

Qu'apporte la modélisation des moteurs en plus de la modélisation du microcontrôleur ? Nous connaissons exactement l'état de notre microcontrôleur grâce à la modélisation de l'impact que les entrées/sorties peuvent produire. Cependant, comme nous l'expliquons ci-après, les moteurs pas à pas ont une subtilité que nous souhaitons intégrer dans notre système de vérification.

L'objectif d'avoir une vérification qui dépend de la spécification physique des moteurs est un aspect novateur. En effet, dans l'industrie il n'est pas rare que de nouveaux projets héritent d'anciens projets. C'est ce qui a poussé Boeing à alléger la certification du 737 Max (cf. §5.2). C'est également ce qui est à l'origine de l'échec du vol 501 de l'Ariane 5. En effet, au vu de la proximité avec Ariane 4, le code a été dupliqué mais pas totalement adapté. Cela a mené à une

exception logicielle qui a conduit à l'autodestruction de la fusée (Robinson 2011).

Afin d'éviter ce genre de situation où le code doit manuellement être vérifié, nous apportons un exemple sur des moteurs pas à pas. Notre objectif est de pouvoir rapidement changer le type de moteur et de laisser F⋆ vérifier la justesse de ce code. Bien sûr, cela ne permet pas d'optimiser le code pour ce nouveau moteur. Toutefois, le programme respectera la spécification du moteur.

Dans cet exemple, nous expliquons comment un moteur pas à pas fonctionne et quelles règles physiques entrent en jeu. Ensuite, nous expliquons comment le programme peut interagir avec son environnement physique. Puis nous rentrons dans les détails de la modélisation. L'objectif de cet exemple est de montrer comment F⋆ peut modéliser et vérifier un système *cyberphysique*.

> Cette section décrit le moteur pas à pas tel que présenté dans (Talpin et al. 2019). Ce travail est un exemple de vérification de système *cyberphysique* qui permet d'illustrer l'utilisation de la chaîne de vérification de F⋆ dans un système qui tient compte des contraintes physiques.

6.1.1 Fonctionnement et réalité physique

Dans cette sous-section, nous décrivons de façon élémentaire le fonctionnement d'un moteur pas à pas. Un tel moteur est composé de deux parties : une partie mobile appelée rotor et une partie fixe appelée stator. Le rotor est aimanté et le stator est équipé de bobines. Nous considérons une version simplifiée équipée d'un stator ayant 4 bobines (appelées A,B,C,D) et d'un rotor de 2 faces. Nous ajoutons qu'il existe des documentations fournissant plus de détails sur le fonctionnement d'un tel moteur (*Stepper Motor Guide* 2020).

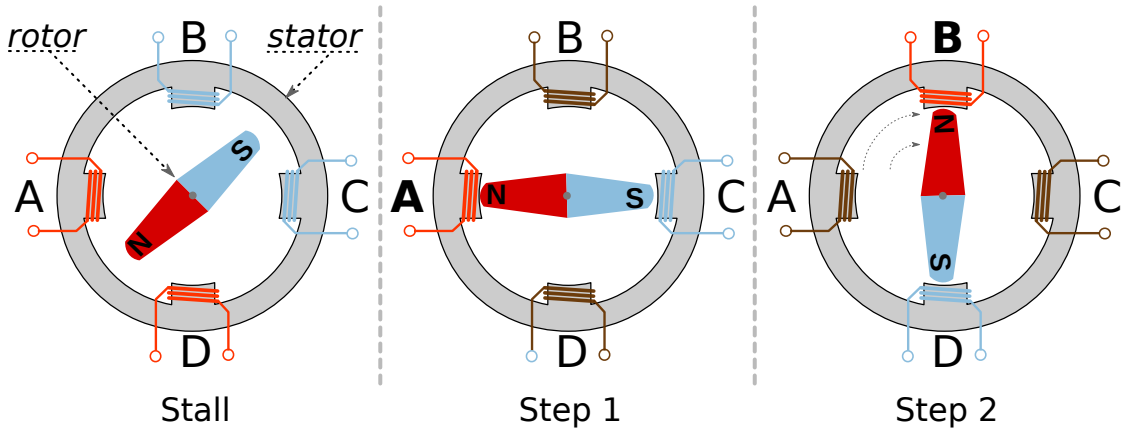

Figure 6.1 – États possibles d'un moteur pas à pas

Initialement, nous éteignons toutes les bobines du moteur sauf une. Dans notre exemple illustré par la figure 6.1, nous dirons que A est cette bobine. Cela fait que le rotor s'aligne donc sur la bobine A dans l'axe A-C. Pour faire le pas suivant, nous éteignons la bobine A et allumons la bobine B. Le rotor s'aligne sur l'axe B-C. Grâce à ce mouvement, nous avons réalisé

un pas. Grâce à cette approche, nous ne sommes pas obligés d'avoir un mécanisme qui mesure le mouvement de notre moteur : nous pouvons simplement compter le nombre de pas effectués.

Maintenant, imaginons que nous allons trop vite. Le rotor peut être attiré par la bobine précédente, puis par la suivante et donc rester bloqué entre deux pas. Cette situation dangereuse appelée *stall* – décrochage en français – (figure 6.1) provoque un blocage. Nous soulignons que la vitesse n'est pas le seul facteur déclencheur. Par exemple, si le rotor est face à une force trop importante le moteur pourrait également être bloqué. Cependant, nous nous focalisons uniquement sur le problème lié à la vitesse.

Une explication plus précise est liée à la force du moteur. En effet, plus le moteur tourne vite moins il est puissant. Ce phénomène vient de l'inductance et du temps pendant lequel les bobines du moteur se chargent. Si le temps entre deux pas est trop court la bobine ne pourra pas se charger complètement, et dans ce cas le moteur verra sa force diminuer. Si la charge est trop importante le moteur décrochera et donc sera bloqué (*What is stepper motor resonance* 2020) (Talpin et al. 2019). Nous nous focalisons sur la vitesse car même avec une charge nulle, le moteur atteindra une limite qu'il ne peut pas dépasser.

Comment faire tourner notre moteur à une vitesse arbitraire ? Cette question est simple, il suffit d'avoir un délai constant entre chaque pas. En fait, nous modélisons notre système comme ayant une vitesse limite et notre moteur comme n'ayant pas d'accélération : il passe instantanément à la vitesse donnée ; la réalité diffère un peu puisque dans un grand nombre d'applications industrielles, l'accélération permet d'augmenter la vitesse limite provoquant le décrochage du moteur. Le code simplifié en C ci-dessous donne l'idée d'un tel programme qui fait tourner le moteur à 40 tours par seconde. La fonction `set_coil` est censée éteindre les autres bobines.

```
coil coils[4] = {A, B, C, D};    /* déclaration des bobines          */
for(int x = 0; x < 4; x++){      /* pour toutes les bobines          */
    set_coil(coils[x]);          /* activation de la bobine courante   */
    delay_ms(25);                /* attente active de 25 millisecondes */
}
```

Bien que simplifié, nous utiliserons cet exemple pour modéliser la bonne utilisation du moteur dans le respect de la spécification matérielle. Comme vu dans 5.2, nous montrons comment la vérification peut être utilisée pour contrer cette faute matérielle.

6.1.2 Modélisation du temps

Dans notre exemple, l'espacement temporel entre deux pas du moteur est la seule variable sur laquelle nous pouvons agir. Nous devons modéliser le temps qui sépare les pas du moteur. En effet, si l'espace temporel qui sépare deux pas est trop court alors le moteur risque d'être bloqué. Grâce à cette représentation spatiale, il nous est facile de raisonner sur le temps puisqu'il suffit de comparer la distance spatiale entre deux pas.

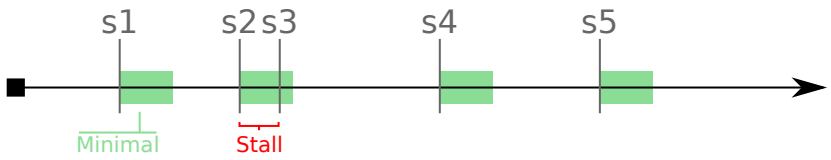

FIGURE 6.2 – Représentation graphique de l'espacement temporel entre deux pas

Nous pouvons représenter le temps par un entier naturel ($\mathbb{N}$) qui augmente au fur à mesure. Ainsi, si un pas est effectué à un instant `t` nous savons que le programme doit attendre au moins un temps minimal `n` et le pas suivant ne doit pas être effectué avant `t+n`. Dans la figure 6.2, nous illustrons comment cette distance permet de détecter les pas qui sont trop rapprochés. Ainsi, les pas `s2` et `s3` sont trop proches et créeront un décrochage du moteur.

Nous pourrions modéliser le temps en ajoutant uniquement un entier naturel dans la définition d'effet. Cependant, nous souhaitons prendre en compte plusieurs moteurs ayant potentiellement des spécifications matérielles différentes, ce qui implique de mémoriser le dernier moment où chaque moteur a effectué un pas. Nous définissons un tel contexte dans le code ci-dessous où `board` est la modélisation de l'Arduino, `lastStepped` est une fonction qui retourne le dernier temps où un moteur a été utilisé et `int` est le temps courant.

Le type `lastStepped` décrit une fonction qui prend un entier représentant le numéro du moteur et retourne optionnellement un entier qui est soit le temps où ce moteur a été utilisé pour la dernière fois, soit `None`. Dans ce dernier cas, le moteur n'a jamais été utilisé. Nous utilisons la même méthode pour la gestion des pattes – décrite à la section 5.2 – afin de stocker le temps de dernière utilisation de chaque moteur.

```
type lastStepped = int → option int

type context = board * (lastStepped * int)
```

Nous faisons remarquer que nous avons défini un couple dont chaque partie est aussi un couple (`board` est un tuple implicite défini dans la section 5.2). On remarquera également que nous utilisons le type `int` qui décrit les entiers relatifs ($\mathbb{Z}$). Cela est dû à une limitation actuelle de F$\star$ qui empêche d'avoir un type raffiné dans la définition du contexte. C'est pourquoi nous n'utilisons pas le type `nat` ($\mathbb{N}$). Toutefois pour nous assurer que raisonner sur des valeurs négatives n'est pas possible, nous définissons un prédicat `wellTyped` qui retourne vrai si le contexte est correctement typé.

```
let wellTyped ctx = let (a,(b,c)) = ctx in c ≥ 0
```

Maintenant que nous avons un contexte, nous pouvons définir un effet appelé `TIME`. Cet effet est en l'état incompatible avec notre effet `STATE` de l'Arduino : nous devons définir comment relier ces deux effets. Nous définissons alors une élévation de notre effet dans la méthode `lift_state_time`. Cette fonction indique quel est le lien entre les deux effets et notamment

comment passer de l'un à l'autre. Comme notre effet est plus expressif puisqu'il contient le temps, nous indiquons à F⋆ comment lier les deux effets. Nous pourrons inclure des fonctions de l'Arduino dans notre effet mais l'inverse ne sera pas possible.

```
total new_effect TIME = S.STATE_h (context)

unfold let lift_state_time (a:Type) (wp:STATE?.wp a) : TIME?.wp a
    = λ (c0:context) (p:TIME?.post a) →
        wp (fst c0) (λ (v,c1) →
                p (v, (c1, (snd c0)))
        )
sub_effect STATE ⇝ TIME = lift_state_time
```

Nous savons que notre effet `TIME` embarque toutes les données de l'effet `STATE`. La fonction `lift_state_time` va simplement enlever les informations relatives à l'effet `TIME` via `fst c0` et restituer ces informations après l'appel d'une fonction dans l'effet `STATE` via `snd c0`. À ce moment, `c1` est la version du futur de `fst c0` : c'est la modélisation de l'Arduino mise à jour après appel de la fonction `STATE`. La création d'un nouveau contexte avec l'état à jour et les informations temporelles permet de restituer les informations nécessaires à l'effet `TIME` : les fonctions dans `STATE` n'ont aucun coût temporel et donc appeler une telle fonction ne change qu'une partie du contexte `TIME`.

De façon très similaire à la section 5.2, nous définissons comment utiliser les pré- et post-conditions dans un effet que nous appelons cette fois `Time`. Toutefois, nous forçons la justesse du typage de notre contexte dans cette définition avec le prédicat `wellTyped`. Il est important de comprendre que cette étape est un palliatif au problème de la définition des effets décrit précédemment. En effet, cette définition diverge de la définition de `TIME` et rien n'empêche le développeur d'utiliser `TIME`. Cependant, le développeur qui rendrait ce système non cohérent en utilisant `TIME` ne pourrait plus appeler les fonctions qui utilisent l'effet `Time`.

```
effect Time (a:Type) (pre: context → Type0)
    (post: context → a → context → Type0)
=
    TIME a (λ n0 p →
        pre n0
        ∧ wellTyped n0
        ∧ (∀ a n1. (
            pre n0
            ∧ post n0 a n1
            ∧ wellTyped n0
            ∧ wellTyped n1) ⟹ p (a, n1)))
```

Cette définition d'effet nous permet d'écrire des pré- et post-conditions de manière plus simple. Nous devons décomposer à chaque fois le contexte mais nous pouvons raisonner sur

l'Arduino et le temps en même temps. Le code ci-dessous illustre une fonction avec les pré- et post-conditions.

```
let foo (_:unit) : Time (unit)
    (requires λ (board,(last,time)) → ⊤)
    (ensures λ (b0,(l0,c0)) x (b1,(l1,c1)) → ⊤)
=
    ()
```

Dans la précondition, nous décomposons un contexte en triplet où `board` est la modélisation de l'Arduino, `last` le stockage du dernier accès des moteurs et `time` le temps courant. De même dans la postcondition, le contexte passé est décomposé en `b0`, `l0` et `c0` et le contexte futur – après l'appel de notre fonction `foo` – est décomposé en `b1`, `l1` et `c1`.

Nous avons vu comment modéliser le temps et l'intégrer dans la modélisation Arduino existante. Maintenant, nous allons voir comment utiliser cette modélisation afin de prévenir un programme qui provoque un décrochage du moteur.

6.1.3 Modélisation d'un moteur

Normalement dans l'industrie, les développeurs prototypent un système en branchant les composants puis développent avec des fonctions standards le code de l'application embarquée. Nous souhaitons que notre vérification du moteur s'approche de l'utilisation des fonctions standards. Nous avons donc simplifié au maximum la déclaration d'un moteur pour être au plus près de l'utilisation naturelle d'une Arduino.

Nous avons vu que nous communiquons avec le monde physique via des pattes dans l'Arduino. C'est de cette manière que nous allons changer l'état des bobines dans notre moteur. Il existe différentes formes de connectique entre les modèles de moteurs pas à pas et dans les modèles courants pour l'Arduino, il en existe deux sortes : les moteurs qui utilisent deux pattes numériques et ceux qui utilisent quatre pattes numériques. Nous allons nous focaliser sur les moteurs à quatre pattes numériques, sachant que nous pouvons appliquer ce que nous présentons dans cette sous-section sur des moteurs à deux pattes.

Comme illustré par la figure 6.3, nous choisissons des pattes numériques auxquelles nous relions un pilote par des fils électriques. Le rôle du pilote est de gérer le moteur. Ce pilote est un microcontrôleur qui est généralement fourni par le constructeur du moteur, c'est une des raisons pour lesquelles notre modélisation ne prend en compte que l'Arduino. En effet, les pilotes peuvent être très variés et dans ce cas nous ne pourrions en vérifier qu'un seul.

Comme nous l'avons vu, la sorte de moteur que nous manipulons utilise quatre pattes. Nous décrivons donc une structure contenant quatre pattes numériques – numérotées de 1 à 4 – qui correspondent aux branchements du moteur sur l'Arduino. Cependant, chaque moteur a une spécification qui dépend de ses caractéristiques. Nous ajoutons alors un champ `minimal_delay` qui correspond au nombre minimal de millisecondes que ce moteur est physiquement capable de

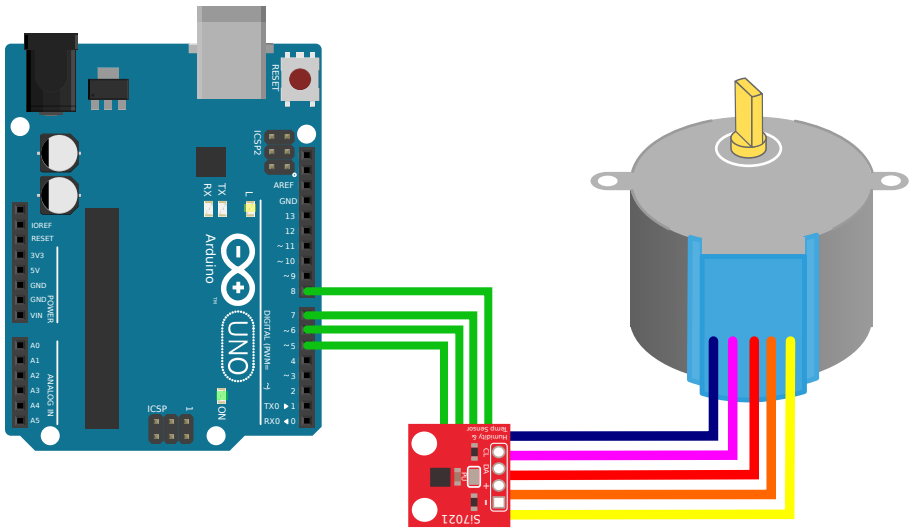

FIGURE 6.3 – Connexion matérielle du moteur à l'Arduino

supporter entre deux pas. N'oublions pas que ce nombre pourrait être augmenté si nous tenions compte de l'accélération.

```
noeq
type motor = {
    minimal_delay:G.erased nat;
    coil1:digital_pin;
    coil2:digital_pin;
    coil3:digital_pin;
    coil4:digital_pin;
}
```

Nous faisons observer que `minimal_delay` est un entier naturel effacé à la compilation ; grâce à `erased`. Cela permet de ne pas extraire ce code en C mais nous oblige à rendre cette structure de données non comparable ; nous le faisons grâce au mot-clef `noeq`.

Dans le code F⋆ précédent, notre moteur est déclaré mais ne tient pas compte d'une contrainte assez simple : chaque patte transmettant une information spécifique, il ne doit normalement pas être possible d'avoir des pattes identiques pour des bobines différentes. Pour spécifier cela, nous ajoutons un prédicat qui retourne vrai pour un moteur respectant ces contraintes. Cela nous permet d'ajouter par raffinement un type plus précis que nous appelons `motor_specification`.

```
let motorWellFormed m =
    let c1,c2,c3,c4 = U8.(v m.coil1, v m.coil2, v m.coil3, v m.coil4) in
    c1 ≠ c2 ∧ c1 ≠ c3 ∧ c1 ≠ c4
    ∧ c2 ≠ c3 ∧  c2 ≠ c4
    ∧ c3 ≠ c4
```

```
type motor_specification = m:motor{motorWellFormed m}
```

Dans la sous-section précédente, nous avons défini un état qui, avec l'identifiant d'un moteur, permet de retourner le temps où il a été utilisé pour la dernière fois. Toutefois la structure ne contient pas d'identifiant. Bien qu'elle le pourrait, nous savons que les moteurs ont toujours des numéros de pattes différents. En effet, si deux moteurs partagent les mêmes pattes, ils sont identiques du point de vue de l'Arduino. Nous générons un identifiant unique pour chaque moteur. Il est important de rappeler que `FStar.Mul.op_Star` est l'opération de multiplication sur les entiers.

```
let getMotorId m =
    let c1,c2,c3,c4 = U8.(v m.coil1, v m.coil2, v m.coil3, v m.coil4) in
    (c1 `FStar.Mul.op_Star` c2 `FStar.Mul.op_Star` c3) + c4
```

Maintenant nous savons comment stocker le temps de dernière utilisation de chaque moteur, le temps global et l'interconnexion entre la modélisation de l'Arduino et la modélisation du temps. Nous allons montrer comment utiliser ces éléments pour vérifier la bonne utilisation des moteurs.

6.1.4 Primitives du moteur

Nous avons vu précédemment que les moteurs fonctionnent en deux étapes : 1/ la configuration des bobines ; 2/ l'attente temporelle. Il est important de rappeler que nous nous focalisons sur l'attente active ; c'est-à-dire que l'Arduino est bloquée pendant ce temps. Il nous est possible d'appliquer la méthode que nous utilisons pour une attente passive. Cependant, dans ce cas, la mise en œuvre est plus complexe.

Dans l'exemple d'utilisation des moteurs nous avons donc deux primitives importantes : `set_coil` qui met le moteur dans la configuration du pas suivant et `delay_ms` qui crée une attente active. Ces deux fonctions permettent d'implémenter les fonctions de la bibliothèque standard.

Pour mettre le moteur dans un pas, nous avons défini la méthode `set_coil` qui prend un `motor_specification` et un numéro de bobine. Cette fonction configure le moteur pour assurer que la bobine sélectionnée est active pendant que les autres ne le sont pas. La fonction présentée ci-après décrit la configuration d'un moteur. Les numéros de bobine sont représentés par le type `coil_number` qui permet de dire quelle bobine doit être active.

Le code ci-dessous nous permet de montrer complètement l'utilisation de notre système de vérification en F⋆. Nous avons deux prédicats, `motorWrite_pre` et `motorWrite_post`, qui vérifient que les pattes numériques sont dans le bon mode pour un moteur donné. La précondition vérifie que si le moteur a été utilisé avant alors il s'est écoulé au moins le délai minimum depuis la dernière utilisation du moteur : `(c - last) ≥ (G.reveal m.minimal_delay)`. La postcondition vérifie que le nouveau temps enregistré dans la modélisation est le temps courant : `getLastStepped l1 (getMotorId m) == Some c1`. Nous faisons remarquer que cette fonction

ne modifie pas le temps courant puisque `c0 == c1`. Il est aussi important de noter que cette fonction modifie les quatre pattes auxquelles est rattaché le moteur. De plus, l'utilisation de `G.reveal` est nécessaire afin d'accéder aux informations effacées – elles ne sont jamais extraites.

Le corps de fonction est composé d'instructions propres à la modélisation et d'instructions propres à l'Arduino. Dans un premier temps, nous configurons les pattes numériques via les `digitalWrite`. Cela permet de modifier physiquement l'état du moteur. Puis nous mettons à jour la modélisation en enregistrant le temps courant pour le moteur passé en paramètre.

```
type coil_number = | Coil1 | Coil2 | Coil3 | Coil4

let set_coil (m:motor_specification) (coil:coil_number) : Time (unit)
    (requires λ (b,(l,c)) →
        board_initialized b
        ∧ motorWrite_pre m b
        ∧ (Some? (getLastStepped l (getMotorId m))
              ⟹ (let Some last = getLastStepped l (getMotorId m) in
             (c - last) ≥ (G.reveal m.minimal_delay))))
    (ensures λ (b0,(l0,c0)) x (b1,(l1,c1)) →
        c1 = c0
        ∧ getLastStepped l1 (getMotorId m) == Some c1
        ∧ motorWrite_post m b0 b1
        ∧ modifies b0 b1 [m.coil1;m.coil2;m.coil3;m.coil4])
=
    digitalWrite m.coil1 (if Coil1? coil then HIGH else LOW);
    digitalWrite m.coil2 (if Coil2? coil then HIGH else LOW);
    digitalWrite m.coil3 (if Coil3? coil then HIGH else LOW);
    digitalWrite m.coil4 (if Coil4? coil then HIGH else LOW);
    let (b1,(l1,c1)) = TIME?.get () in
    TIME?.put (b1,(setLastStepped l1 (getMotorId m) c1,c1))
```

Grâce à la vérification dans F⋆, nous savons que le corps de notre fonction correspond à la modélisation que nous avons décrite puisque le code doit concorder avec la spécification. Cependant, nous pouvons observer que nous ne disons pas à F⋆ que seule une bobine doit être active. Pour ajouter cette information dans notre modélisation, nous écrivons le prédicat suivant :

```
let onlyOneCoil b m coil =
    checkValue b m.coil1 (if Coil1? coil then HIGH else LOW)
    ∧ checkValue b m.coil2 (if Coil2? coil then HIGH else LOW)
    ∧ checkValue b m.coil3 (if Coil3? coil then HIGH else LOW)
    ∧ checkValue b m.coil4 (if Coil4? coil then HIGH else LOW)
```

Ce prédicat dit que toutes les bobines sont `LOW` – c'est-à-dire inactives – à l'exception de celle donnée en paramètre qui est `HIGH`. En rajoutant ce prédicat dans la postcondition, nous

pouvons vérifier que notre fonction `set_coil` se comporte comme souhaité. Nous avons également prouvé que le comportement de `onlyOneCoil` est correct. Cette preuve est une simple preuve par énumération pour chaque bobine.

Maintenant que nous avons montré comment réaliser un pas, nous présentons comment attendre activement entre deux pas. Dans l'Arduino, il existe une fonction appelée `sleep` qui attend le nombre de millisecondes donné en paramètre. Cette fonction est implémentée de la même manière que les primitives de l'Arduino. Cette fonction prend un entier et met à jour le temps courant sans toucher ni à la modélisation de l'Arduino ni à la liste des dernières utilisations des moteurs.

```
let sleep (d:U16.t) : Time (unit)
    (requires λ (b,(l,c)) → board_initialized b)
    (ensures λ (b0,(l0,c0)) x (b1,(l1,c1)) →
        b0 == b1
        ∧ c1 = c0 + U16.v d
        ∧ l0 == l1)
=
    let (b,(l,c)) = TIME?.get () in
    TIME?.put (b,(l,c + U16.v d))
```

Grâce à l'expressivité de F⋆, cette fonction est composable, ce qui veut dire qu'elle peut être utilisée plusieurs fois et seulement le total du temps écoulé sera pris en compte. Vus depuis un regard d'initié à la programmation embarquée, nous savons que la mesure du temps est imparfaite. Dans un cadre de vérification *cyberphysique*, nous pourrions utiliser un intervalle du temps au plus tôt et au plus tard en fonction de l'imperfection décrite par le constructeur.

6.1.5 Mise en situation

Imaginons que nous ayons un robot équipé de deux moteurs : le premier, `small`, qui peut faire un pas toutes les 100 ms et le second, `big`, qui peut faire un pas toutes les 150 ms. L'extrait suivant montre le code d'une fonction dans le programme :

```
let small  : motor_specification = {
    minimal_delay=100;
    coil1=1uy; coil2=2uy; coil3=3uy; coil4=4uy;
}
let big    : motor_specification = {
    minimal_delay=150;
    coil1=5uy; coil2=6uy; coil3=7uy; coil4=9uy;
}
...
set_coil big   Coil1;
set_coil small Coil1;
```

```
sleep 100us;
set_coil small Coil2;
sleep 50us;
set_coil big   Coil2;
```

Les deux premières lignes positionnent les moteurs sur la première bobine. Ensuite nous attendons 100 ms avant de passer à la seconde bobine de `small`. Enfin 50 ms plus tard – soit 150 ms au total – nous passons à la seconde bobine de `big`. Ce code est correct par rapport à la spécification des moteurs.

Cependant, un nouveau modèle de robot est développé. Dans ce nouveau modèle le moteur `big` est remplacé par un moteur plus lent qui nécessite un délai de 226 ms. Notre code n'est plus valide pour cette nouvelle configuration. En changeant la spécification de `big`, F⋆ rejette l'appel `set_coil big Coil2`. En effet, l'attente cumulée est toujours de 150 ms. En allongeant l'attente, le code est validé par F⋆. Nous donnons la version corrigée ci-dessous :

```
set_coil big   Coil1;
set_coil small Coil1;

sleep 100us;
set_coil small Coil2;
sleep 126us;            (* correction *)
set_coil big   Coil2;
```

Notons que cette correction rend le code correct du point de vue de la spécification du moteur, mais pas du point de vue de son utilisation en fonction de sa fonction dans un système embarqué. Nous n'avons volontairement pas décrit la manière dont les moteurs sont utilisés afin de ne décrire que l'aspect lié à la vérification. Dans une situation réaliste, la modification que nous avons appliquée pourrait poser des problèmes supplémentaires.

6.2 Gestion de mémoire virtuelle

En informatique, la mémoire, on le sait, est une ressource très importante. Comme nous l'avons vu dans la section 4.4, nous devons faire attention à la manière dont cette ressource est gérée au sein d'un programme : un programme est souvent limité par la capacité du matériel sur lequel il s'exécute. Historiquement les premiers ordinateurs exécutaient un seul programme à la fois. Aujourd'hui cette époque est révolue puisque les ordinateurs fonctionnent avec de nombreux programmes qui s'exécutent simultanément sur la même machine.

L'informatique embarquée fait figure d'exception à cause de ses contraintes. En effet, de nombreux microcontrôleurs n'exécutent qu'un seul programme qui utilise toutes les ressources. Cependant, les microcontrôleurs sont de plus en plus puissants et leurs programmes se complexifient. Cela conduit à enrichir les programmes de ces microcontrôleurs avec des fonctionnalités

plus expressives. Une de ces fonctionnalités que nous présentons est la virtualisation de mémoire.

En fait, lorsqu'un programme utilise de la mémoire, il va utiliser des adresses spécifiques ; souvent leurs valeurs sont choisies par le compilateur à la place du développeur. Cependant si un système comporte au moins deux programmes, comment faire si l'un lit et écrit à une adresse donnée et que l'autre souhaite également y avoir accès ? Dans ce cas, nous pouvons apporter plusieurs réponses : 1/ rien n'est fait et les modifications de chaque programme impactent totalement le reste du système, 2/ nous empêchons toute exécution de programme en simultané, 3/ nous adaptons toutes les adresses ou 4/ nous virtualisons la mémoire afin de faire croire à chaque programme qu'il est seul dans la mémoire. Les deux premières solutions ne conviennent pas du tout à ce que nous voulons. La troisième est peu flexible puisqu'elle limite la mémoire utilisable par chaque programme – la mémoire disponible est divisée par le nombre de programmes – mais fonctionne en principe. Cependant, elle nécessite de modifier les programmes en conséquence, ce qui peut être contraignant. La dernière solution permet de résoudre le problème en gardant l'exécution simultanée de programmes et assure une flexibilité sur l'utilisation de la mémoire. Toutefois, cette solution a un coût non négligeable sur les accès en mémoire. Nous verrons que cette approche est flexible, quant au nombre de programmes et à la quantité de mémoire gérée.

Dans l'histoire de l'informatique ce problème existe depuis longtemps et la solution de la virtualisation a été retenue. De plus, elle permet de rester rétro-compatible avec des programmes précédemment compilés. En effet, cette méthode est indépendante du compilateur puisque chaque programme sera lancé comme s'il était seul en mémoire. Ainsi chaque programme commencera par utiliser les premières adresses sans tenir compte des autres programmes. La virtualisation empêchera toute collision entre les programmes. Cela simplifie à la fois la conception du compilateur et du système d'exploitation.

La résolution des adresses virtualisées passe par un composant essentiel de la sécurité informatique : le MMU. Ce composant est la base de la séparation entre le noyau et le domaine applicatif. Une fois configuré par le noyau, il virtualise tous les programmes applicatifs, leur permet d'utiliser beaucoup plus de mémoire que disponible, les confine entre eux et protège le noyau de l'impact des programmes.

L'aspect critique de ce composant en fait une cible de choix pour les attaquants. En effet, de nombreuses attaques utilisent des failles de sécurité afin d'accéder à la mémoire pour lire des informations gardées secrètes par le noyau comme le mot de passe administrateur par exemple. En réalité, lorsqu'un attaquant atteint ce niveau d'accès, il peut tout simplement réécrire l'intégralité du système comme bon lui semble.

Ces dernières années ont montré à quel point ces attaques sont sensibles mais également possibles. Décrite dans l'article (Kuzminykh et Yevdokymenko 2019), l'attaque *Lojax* cible un composant du Basic Input/Output System (BIOS) afin de contourner toutes les mesures de confinement imposées par le système d'exploitation et appliquées par le MMU matériel. Cette attaque permet à un attaquant de prendre le contrôle complet de la machine et résiste à un chan-

gement de système d'exploitation ou de disque dur.

D'autres attaques matérielles comme (Kocher et al. 2019) et (Lipp et al. 2018) montrent l'impact dévastateur que ce type de faille peut apporter. En effet, ces canaux cachés peuvent permettre d'accéder à l'intégralité de la mémoire système – mot de passe ou clef privée par exemple – ou à la mémoire d'autres programmes. Dans le cas de l'attaque Spectre (Kocher et al. 2019), la lecture complète de la mémoire peut se faire via le code javascript d'une page web par exemple. La simple visite d'un site web ou la lecture d'une image vectorielle peut déclencher l'attaque. Cela permet à un attaquant d'espionner très facilement sa victime et donc de faciliter l'élévation de ses privilèges.

Nous avons donné un aperçu de l'importance du confinement entre les programmes, pourtant nous pouvons nous demander pourquoi ce genre de système est si peu développé dans l'informatique embarquée. Comme nous l'avons précédemment évoqué, ajouter des composants matériels peut avoir un coût à la fois énergétique et financier. De plus, pendant des décennies les microcontrôleurs ressemblaient à de simples automates plus faciles à manipuler que des automates industriels. Puis la miniaturisation des composants a permis d'embarquer dans ces microcontrôleurs de plus en plus de fonctionnalités. Il est donc évident que dans le futur – probablement dans un avenir proche avec l'architecture de processeur Reduced Instruction Set Computer (RISC) V – ce genre de fonctionnalités deviendra monnaie courante dans l'informatique embarquée.

Pourtant vérifier un système logiciel de virtualisation de la mémoire peut s'avérer utile. Outre le fait qu'il puisse servir de base pour vérifier un système matériel, il peut surtout combler le manque de sécurité des processeurs déjà existants et permettre aux nouveaux modèles qui ne peuvent pas posséder de MMU matériel d'avoir une sécurité supplémentaire. De plus, un MMU aura toujours un coût sur les performances mais notre vérification pourrait permettre de réduire ce coût pour une application donnée grâce à la modélisation.

> Notre travail étend un MMU logiciel décrit dans (Choudhuri et Givargis 2005) qui permet d'apporter une virtualisation de la mémoire dans les systèmes embarqués sans MMU matériel. Nous apportons un MMU qui, intégré dans un programme F⋆, peut modéliser le comportement de chaque écriture ou lecture en mémoire. Cela nous permet de pouvoir vérifier en amont la justesse d'un système et en identifiant les tests redondants, d'alléger l'impact de ce MMU logiciel.

Dans cette section, nous présentons le fonctionnement d'un système logiciel de gestion de mémoire basique puis nous donnons des détails sur ce qu'il modélise et sur ses fonctionnalités.

6.2.1 Fonctionnement d'un MMU

Afin d'expliquer le fonctionnement d'un MMU, nous allons d'abord expliquer le fonctionnement de la mémoire. La mémoire peut être représentée comme un ruban contenant un octet

par case. Le début commence à l'adresse 0 – appelée `NULL` dans le milieu de la programmation système – et la fin dépend de la capacité du processeur. Si dans un processeur d'ordinateur de bureau la limite théorique peut être d'un adressage maximal 2^{43}, ce qui permet d'avoir plus de 64 Go de mémoire vive, dans un microcontrôleur ce maximum d'adressage est souvent inférieur à 2^{16}, ce qui ne permet pas d'avoir plus de 16 Ko de mémoire vive.

Imaginons la situation suivante : nous avons un microcontrôleur avec 8 Ko de mémoire vive. Nous pouvons partitionner la mémoire en pages de taille de 1 Ko. Nous laissons 1 Ko pour le système et nous pouvons exécuter un programme de 7 Ko. Comment faire si nous voulons en exécuter deux de 3 Ko ? Comme indiqué dans la partie A de la figure 6.4, la réponse est simple : nous virtualisons les adresses de manière à ce que les programmes n'entrent pas en collision. Nos programmes peuvent donc n'utiliser que 3 Ko de mémoire. Cependant, comment faire si nous voulons exécuter plus de programmes ou si nous voulons exécuter des programmes plus lourds ?

La réponse est d'utiliser le *swap* – c'est-à-dire l'échange de mémoire – comme nous le montrons dans la partie B de la figure 6.4. Dans ce cas, nos programmes peuvent utiliser 7 Ko de mémoire chacun. Dans ce dernier cas, sous réserve d'avoir un stockage secondaire, un MMU nous permettra d'augmenter la capacité de notre système à la fois en termes de nombre de programmes et en termes de mémoire. Nous pourrions avec ce système adresser bien plus que 7 Ko.

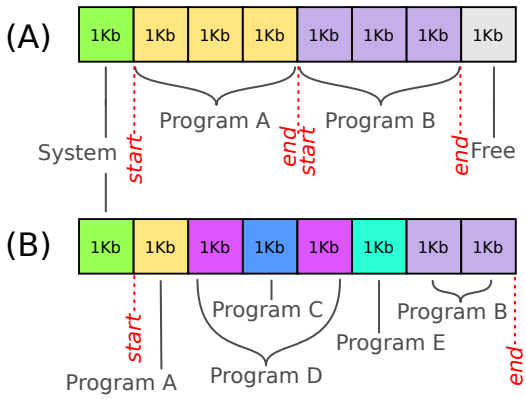

FIGURE 6.4 – Exemple de partitionnement de la mémoire

En effet, de la même manière qu'il nous est possible d'exécuter plusieurs programmes, l'espace de stockage secondaire nous permet théoriquement d'exécuter un programme qui nécessite plus de mémoire. Dans notre exemple, nous pourrions exécuter un programme qui consomme 32 Ko de mémoire bien que notre système n'en possède que 7 Ko de libre. Toutefois, notre système se cantonne à utiliser uniquement l'espace mémoire disponible.

Notons que la figure 6.4 représente un état de la mémoire à un instant donné. En réalité, la répartition des pages dépend de l'exécution courante d'un programme. Lorsqu'un programme souhaite lire ou écrire une valeur en mémoire, le MMU logiciel intercepte cet appel puis calcule l'adresse de la page virtuelle. Il y a alors deux possibilités : soit la page est déjà chargée et la mise à jour de la mémoire est appliquée, soit la page n'est pas chargée – la page courante appartient

à un autre programme – et le MMU va éventuellement enregistrer cette page courante puis il va charger la page du programme ayant demandé l'accès à la mémoire. Nous décrivons en détail l'algorithme dans la sous-section suivante.

Comme pour d'autres exemples, notre code est d'abord écrit en F⋆, ce qui nous permet de modéliser finement l'exécution avant l'extraction en C. Dans une implémentation en C d'un petit système embarqué, nous pourrions facilement faire des erreurs et c'est la raison pour laquelle nous devrions faire régulièrement des tests pendant le développement en phase de débogage mais également à l'exécution sur une application en phase de production. Cependant, nous avons montré que nous pouvions utiliser la vérification pour supprimer des tests car certains sont redondants.

À ce stade, notre objectif est double : 1/ vérifier le bon fonctionnement de notre MMU et 2/ modéliser suffisamment l'état de notre MMU pour tenir compte de l'état de la mémoire afin de réduire les traductions d'adresses virtuelles en adresses réelles. Ce dernier objectif permettrait d'augmenter les performances de notre MMU qui, dans certains cas, pourrait même être plus rapide qu'un MMU matériel. En effet, si un programme vérifié dans un environnement déterministe utilise des adresses constantes, une fois calculés les emplacements mémoire n'ont pas à être recalculés et les opérations de *swap* peuvent être factorisées.

Maintenant que nous avons présenté le rôle du MMU ainsi que nos objectifs, nous présentons notre implémentation ainsi que la modélisation que nous utilisons et ce qu'elle vérifie.

6.2.2 Opérations sur la mémoire virtuelle

Notre algorithme de traduction des adresses est basé sur un MMU logiciel existant (Choudhuri et Givargis 2005). Initialement ce MMU a été conçu pour "patcher" des programmes écrits en C afin d'introduire un confinement sur les microcontrôleurs n'ayant pas de MMU. L'algorithme que nous utilisons pour la traduction des adresses est le même mais nous l'utilisons dans des programmes en F⋆. La figure 6.5 décrit son fonctionnement. Cette fonction que nous appelons `translation` prend une adresse virtualisée `va` et un booléen `writing` qui est vrai si l'action est une écriture et faux sinon.

La première opération consiste à décomposer l'adresse pour extraire les informations concernant sa page `page` et sa position en mémoire `offset`, mais également la tâche à laquelle elle appartient `tag`. Puis l'état correspondant à la page est lu via la fonction `readStatus`. Cette fonction retourne l'état de la page actuellement chargée en mémoire. Ensuite deux cas se présentent :

1. Si cette page appartient à notre tâche (`ctag = tag`) et qu'elle a au moins été utilisée une fois, nous retournons l'adresse réelle. Nous mettons à jour, via `writeStatus`, l'état de la page, seulement si nous allons altérer la page avec une écriture. Cette mise à jour indique que le contenu de la page n'est plus synchronisé avec le contenu de la page dans le stockage secondaire.
2. Si cette page n'a jamais été utilisée ou qu'elle appartient à une autre tâche, nous ne

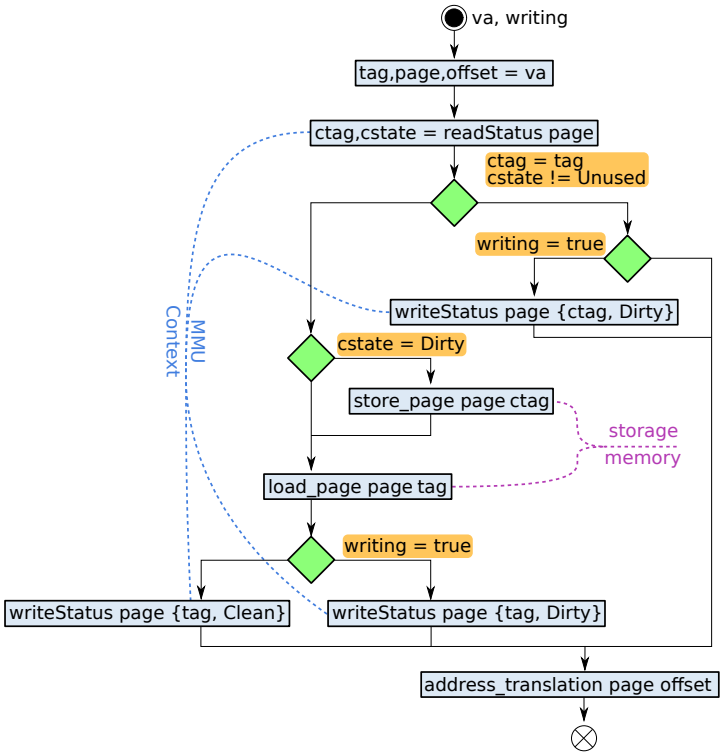

FIGURE 6.5 – Représentation graphique de l'algorithme de traduction des adresses

sauvegardons la page courante que si elle n'est pas identique à sa version dans le stockage secondaire. Ensuite, nous chargeons la page appartenant à notre tâche depuis le stockage secondaire. Puis nous mettons à jour cette page courante à `clean` ou `dirty` selon que la page est synchronisée avec le stockage secondaire ou pas.

La dernière étape retourne une adresse physique via la fonction `address_translation`. Cette adresse physique est la position dans la mémoire réelle pointée par l'adresse virtuelle. Le MMU ayant synchronisé le contenu en mémoire, les données appartiendront à la mémoire de la tâche demandée.

Notons que n'importe quel programme peut accéder à la mémoire. Techniquement en l'état c'est vrai. Toutefois, nous montrerons plus loin (voir le chapitre 7) comment nous en prémunir.

6.2.3 Description de la mémoire virtuelle

Le contexte de notre MMU doit contenir en mémoire le nombre de tâches qu'il peut au maximum exécuter, mais il doit en plus savoir à quelle tâche appartient une page donnée et quel est son état. Une page appartient à une tâche identifiée par `tag` et par son état. Il n'y a que trois états possibles : `Unused` qui indique que la page n'est pas utilisée, `Clean` qui précise que la page est chargée et ses données sont synchronisées avec le stockage secondaire et `Dirty` qui indique que la page est chargée mais que les données ne sont plus synchronisées avec le stockage secondaire. Nous donnons la déclaration du code F⋆ ci-dessous :

```
type page_state = | Unused | Clean | Dirty

type table_page_entry = {
    tag     : task_tag;
    state   : page_state;
}
```

Contrairement à la modélisation de l'Arduino, le contexte va être conservé dans le code extrait. Cela permet d'avoir un MMU dynamique ce qui est dans notre cas nécessaire puisque nous souhaitons que les programmes puissent utiliser dynamiquement de la mémoire sans passer par le modèle de L⋆. Nous soulignons que nous avons volontairement utilisé un "contournement" dangereux dans notre contexte : le champ `tpe` est un tableau d'une taille non définie. Cela nous permet d'extraire le tableau `tpe` comme un tableau normal en C et donc de permettre à nos abstractions de la mémoire de fonctionner correctement.

> Nous n'utilisons pas le modèle mémoire de L⋆ pour deux raisons : la première vient de l'état de développement de F⋆ au moment de ce travail, qui ne permettait pas de combiner des effets, ce qui est maintenant possible grâce aux *layered-effect* ; la seconde raison est qu'un effet mémoire performant devrait prendre en compte les données stockées sur la pile mémoire et cela nécessite des changements importants qui ne nous ont pas paru réalisables dans le temps imparti. Ce MMU est donc un exemple jouet.

Toutefois, pour éviter d'avoir des problèmes de cohérence, nous modélisons le comportement de ce tableau par une version réfléchie appelée `tpe_reflection`. Cette réflexion est similaire à celle décrite dans la section 5.2 où nous expliquons comment fonctionne la réflexion des entrées/sorties. Dans le code ci-dessous, nous décrivons un contexte contenant les informations suivantes : l'état de l'initialisation, le nombre de tâches prises en compte (ici nous aurons toujours quatre tâches), un tableau de `table_page_entry` et la version réfléchie de ce tableau.

```
noeq
type context_state = {
    initialized:mmu_state;
    nb_process:(b:byte{Byte.v b = 4});
    tpe:B.buffer (table_page_entry);
    tpe_reflection:G.erased (reflected_tpe nb_process);
}
```

Notre MMU doit également être initialisé. En effet, il peut être dans trois états : l'initialisation de la mémoire à utiliser, l'initialisation du contexte et l'état de fonctionnement. Nous faisons remarquer que nous aurions pu rajouter un état pour indiquer une exception mais nous ne sommes pas allés aussi loin.

Le modèle mémoire que nous utilisons est proche de celui du langage C : il peut être vu comme un ruban. Bien que nous ayons décrit comment utiliser la mémoire en F⋆ dans la section 4.4, nous n'allons pas utiliser cette approche afin de simplifier notre implémentation. Dans notre exemple de MMU, nous utilisons des adresses 32 bits mais l'adaptation vers des adresses plus petites ne devrait pas être compliquée. En effet, le code a été pensé pour être flexible à ce changement ; les adresses sont définies de manière à pouvoir être rapidement changées. Afin de vérifier finement la justesse de ces accès, nous avons défini des types raffinés par la configuration du MMU. Comme nous le décrivons dans les sous-sections précédentes, nous utilisons une mémoire finie découpée en pages. Ce découpage est très commun dans la programmation système.

En utilisant l'expressivité de F⋆ et en s'inspirant des lemmes de la bibliothèque standard, nous laissons le développeur choisir les constantes qu'il souhaite tout en refusant celles qui ne respectent pas la spécification. Pour cela, nous définissons la constante `max_memory` qui fixe la taille de la mémoire que notre MMU peut gérer. Nous précisons par la propriété `Addr.v a > 0` que cette taille doit être supérieure à 0. La page est également définie par raffinement. Notons que le module `Addr` est un renommage du module `FStar.UInt32`.

Cependant, la taille d'une page est dépendante de la taille de la mémoire puisqu'elle la divise. Nous le précisons par la propriété `(Addr.v max_memory % Addr.v a) = 0` dans la constante calculée `page_size`. Nous ajoutons que la taille d'une page ne peut pas être plus grande que la mémoire par `Addr.v a < Addr.v max_memory`. Et via `page_number`, le nombre de pages est une constante calculée dont le raffinement assure que la valeur représente bien le nombre de pages dans notre mémoire. Ainsi choisir des valeurs erronées dans la configuration provoquera une cascade d'échecs dans la vérification.

```
module Addr = FStar.UInt32
type address = Addr.t

let max_memory : a:address{Addr.v a > 0}
= 32768ul

let page_size : a:address{
    Addr.v a > 0
    ∧ Addr.v a < Addr.v max_memory
    ∧ (Addr.v max_memory / Addr.v a) > 0
    ∧ (Addr.v max_memory % Addr.v a) = 0}
= 512ul

let page_number :  (a:address{
    Addr.v a > 0
    ∧ Addr.v a = Addr.v max_memory / Addr.v page_size
    ∧(Addr.v a `Mul.op_Star` Addr.v page_size) = Addr.v max_memory})
```

```
= max_memory `Addr.div` page_size
```

Grâce à ces définitions, F⋆ validera une mémoire de taille 32 Ko avec des pages de 512 octets. Par exemple si le développeur se trompe et écrit 521 octets en inversant deux chiffres, cette vérification échouera. Il est important de souligner que cette vérification n'est pas dynamique ; le code ne contiendra que les valeurs, car les propriétés sont vérifiées à l'extraction et ensuite supprimées.

Pour cet exemple, nous définissons la taille du stockage secondaire comme un multiple de la taille de la mémoire par le nombre de tâches que notre système possédera. Ceci est une simplification que nous faisons volontairement. Le code ci-dessous calcule la taille du stockage secondaire. Nous faisons remarquer que cette fonction prend un contexte puisqu'elle est dépendante du nombre de tâches dans le système. Pour simplifier, si notre système a 4 tâches (la valeur de `nb_process` sera égale à 4) alors la taille du stockage secondaire sera 4 fois plus grande que la taille de la mémoire. Cette fonction a deux objectifs : calculer correctement la taille du stockage secondaire et enrichir la valeur retournée d'un type raffiné.

```
let max_storage (c:context_state) : (a:address{
    Addr.v a ≥ Addr.v max_memory
    ∧ (Addr.v a / Addr.v max_memory) = Byte.v c.nb_process
    ∧ (Addr.v a % Addr.v max_memory) = 0})
= max_memory `Addr.mul` (C.uint8_to_uint32 c.nb_process)
```

Grâce à nos fonctions, nous pouvons déclarer trois types raffinés : un pour décrire les adresses virtuelles appelé `virtual_address`, un pour décrire les adresses réelles de la mémoire gérée par le MMU que nous appelons `physical_address`, et enfin un type pour décrire les adresses réelles dans le stockage secondaire que nous nommons `storage_address`.

```
type physical_address = a:address{Addr.v a < Addr.v max_memory}
type storage_address (c:context_state) = a:address{Addr.v a < Addr.v (
    max_storage c)}

type tpe_index = a:address{Addr.v a < Addr.v page_number}

type offset_index = a:address{Addr.v a < Addr.v page_size}

type virtual_address = {
    vatag : tag;          (* task id *)
    page:tpe_index;       (* page number *)
    offset:offset_index (* position in page *)
}
```

Maintenant que nous avons introduit différents types, nous allons montrer comment fonctionne notre fonction de traduction des adresses virtuelles en adresses réelles. Dans notre exem-

ple précédent, nous avions un système avec 8 Ko de mémoire. Dans la figure 6.6, nous représentons cette mémoire en superposant les adressages réels et virtuels.

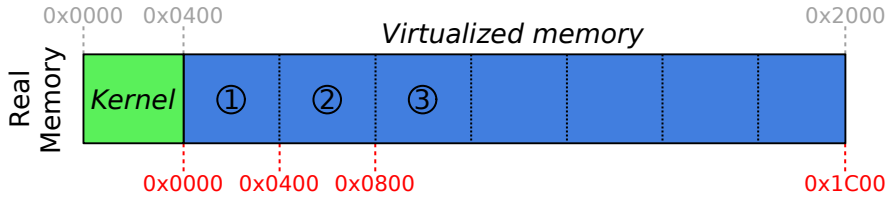

FIGURE 6.6 – Superposition des adresses réelles et virtuelles

Imaginons un exemple où nous souhaitons écrire dans la seconde page à la position 5. Nous calculons la position dans la mémoire virtuelle en tenant compte de la taille des pages en multipliant l'index de la page pi avec la taille d'une page ps puis en ajoutant la position os. Cette formule, $pi * ps + os$, est facilement vérifiable puisque les types définis précédemment empêchent d'exprimer des valeurs incorrectes pour les variables. Si nous utilisons un système embarqué nous devons protéger le code de notre programme en décalant le début de la mémoire gérée après l'espace réservé – dans notre exemple, il s'agit de 1 Ko (`0x400`).

Dans notre exemple, accéder au cinquième octet de la seconde page nous donne `0x400 + (0x1 * 0x400 + 0x5)` soit `0x805`. Quiconque habitué à la manipulation de mémoire remarquera que la seconde page est identifiée par l'index 1 et que la taille d'une page correspond à 1024 octets soit `0x400` en hexadécimal. Dans nos expérimentations sur machine, nous utilisons comme adresse de départ celle d'un tableau en mémoire pour simuler la mémoire.

Notre algorithme de traduction d'adresses en F⋆ est une transcription directe de notre calcul sauf que nous ne protégeons pas le code du programme puisque le calcul devrait dépendre de la taille du programme ainsi que de l'architecture. Nous expliquons plus loin que nous utilisons des primitives en C pour accéder à la mémoire et c'est dans ces fonctions que nous déléguons cette protection puisque ces fonctions sont adaptées à l'architecture ciblée. Nous donnons le code de la fonction ci-dessous. Nous faisons observer que nous utilisons un lemme. En fait, F⋆ échoue à prouver automatiquement que les adresses calculées soient bien comprises dans l'espace d'adressage défini par le type `physical_address`.

```
let translate (page:tpe_index) (offset:offset_index) : Tot physical_address
=
    inbound_lemma   (Addr.v max_memory)
            (Addr.v page_size)
            (Addr.v page_number)
            (Addr.v page) (Addr.v offset);
    (page `Addr.mul` page_size) `Addr.add` offset
```

Bien que ce code soit simple, nous devons parfois guider F⋆ afin de décomposer un problème en plus petits morceaux. La propriété que F⋆ a du mal à vérifier est que $pi * ps + os$ est inférieur

à la mémoire gérée mm. Nous lui demandons de prouver que toutes les pages sont comprises dans la mémoire : $ps * pn \leq mm$. Cela est facile puisque F⋆ suit la définition des types. Nous demandons alors de vérifier que le début de la dernière page ($pn - 1$) plus la position (os) est bien inférieur à la mémoire gérée, ce que F⋆ prouve. En utilisant ce lemme dans la fonction `translate`, nous rejouons cette preuve et permettons ainsi à F⋆ de valider notre fonction.

```
let inbound_lemma (mm:nat{mm > 0})
    (ps:nat{ps < mm ∧ ps > 0})
    (pn:nat{pn = mm / ps})
    (pi:nat{pi < pn})
    (os:nat{os < ps})
: Lemma (ensures(pi `Mul.op_Star` ps) + os < mm)
=
    assert(ps  `Mul.op_Star` pn ≤ mm);
    assert(ps  `Mul.op_Star` (pn - 1) ≤ mm - os)
```

Ce lemme nous permet de prouver que toutes les adresses traduites, dans la fonction de traduction, sont par construction toujours valides dans notre implémentation : il n'est pas possible de créer une adresse qui sort de l'espace d'adressage.

Pour accéder aux mémoires, nous utilisons plusieurs primitives. Deux d'entre elles s'appellent `read_memory` et `write_memory` et permettent respectivement de lire et écrire dans la mémoire. Ces fonctions utilisent des adresses réelles `physical_address`. Deux autres s'appellent `read_storage` et `write_storage` et permettent respectivement de lire et écrire dans le stockage secondaire. Elles utilisent des adresses de la mémoire secondaire appelées `storage_address`.

```
assume val read_memory  : physical_address → MMU_primitives byte
assume val write_memory : physical_address → byte → MMU_primitives unit

assume val read_storage  : (#c:context_state)
    → storage_address c
    → MMU_primitives byte
assume val write_storage : (#c:context_state)
    → storage_address c
    → byte
    → MMU_primitives unit
```

Nous soulignons que ces fonctions ne sont pas vérifiées. En effet, c'est la manière dont elles sont appelées que nous vérifions. Nous pouvons illustrer la façon dont ces fonctions sont utilisées dans la fonction `translation`. Ce code F⋆ est l'implémentation de l'algorithme présenté sur la figure 6.5. Dans ce code, nous montrons comment ces fonctions de manipulation de la mémoire sont utilisées.

```
let translation (va:virtual_address) (writing:bool) : MMU_Safe (
   physical_address)
```

```
    (requires λ c → tpe_allocated c.tpe_reflection va.page)
    (ensures λ c0 x c1 →
        (let {vatag=tag; page=page; offset=offset} = va in
        (tpe_allocated c1.tpe_reflection va.page)
        ∧ (modifies_only c0 c1 [page])
    ))
=
    let {vatag=tag; page=page; offset=offset} = va in
    let pte : table_page_entry = readStatus page in
    if (pte.tag = tag) && not (Unused? pte.state) then (
        (if writing then
            writeStatus page ({pte with state=Dirty}) else ());
        translate page page_size offset
    ) else (
        (if (Dirty? pte.state) then
            (store_page page) else ());
        (load_page page tag);

        (if writing then
            writeStatus page ({pte with state=Dirty})
        else
            writeStatus page ({tag=tag; state=Clean}));
        translate page page_size offset
    )
```

Nous précisons que nous vérifions peu de choses : nous vérifions que la table de pages est bien utilisée et qu'une seule page est mise à jour. En fait, cet exemple n'est pas complet puisque les entrées/sorties sont peu spécifiées : nous ne vérifions que le bon partitionnement de la mémoire. Si nous regardons le code de `write`, cela est flagrant : si l'appel à `write_memory` est incorrect alors notre programme est valide sans pour autant être sémantiquement correct.

```
let write (task:tag) (data:byte) p o : MMU_Safe (unit)
    (requires λ c0 → tpe_allocated c0.tpe_reflection p)
    (ensures λ c0 x c1 →
        (tpe_allocated c1.tpe_reflection p)
        ∧ (modifies_only c0 c1 [p]))
=
    let va = {vatag=task; page=p; offset=o} in  (* décomposition *)
    let real = translation va true in           (* traduction *)
    (write_memory real data)                    (* écriture *)
```

Bien que cet exemple n'ait pas vocation à faire de la virtualisation complètement vérifiée, il permet d'énoncer une limite de la vérification formelle : la précision de la modélisation. En

effet, plus elle sera précise moins il sera possible d'exprimer de programmes, principalement ceux qui ne sont pas réellement corrects ce qui est notre but. Cependant, si cette modélisation est limitée, elle laissera passer des programmes qui sont erronés. La modélisation définit ce qui est considéré comme erroné et cette définition peut risquer d'être en décalage avec les besoins réels du programme. C'est pourquoi le développeur est censé définir une modélisation complète, ce qui n'est pas forcément un problème facile, car une modélisation devient difficile à exprimer en fonction de la complexité du système. De plus, une modélisation incomplète ne décrit que partiellement le système, ce qui ne permet qu'une vérification partielle.

6.2.4 Exemple de programmes avec virtualisation

Pour illustrer comment peut s'utiliser la virtualisation de notre mémoire, nous considérons deux tâches `A` et `B` communiquant par le biais d'adresses mémoire. Ces tâches sont dépendantes de l'ordre dans lequel elles sont exécutées. La première déclare deux caractères de valeur `11` et `22` qui sont stockés dans la mémoire de la tâche `A` aux adresses `0000` et `0001`. Le code ci-dessous en est l'implémentation :

```
let taskA () : MMU_Safe (unit)
    (requires λ c0 → tpe_allocated c0.tpe_reflection 0ul )
    (ensures λ c0 x c1 →
        (tpe_allocated c1.tpe_reflection 0ul)
        ∧ (modifies_only c0 c1 [0ul]))
=
    write A 11uy 0ul 0ul;
    write A 22uy 0ul 1ul
```

Une fois la tâche `A` exécutée, sa mémoire contiendra les valeurs nécessaires pour la tâche `B` qui effectue l'addition et stocke le résultat dans sa mémoire à l'adresse `0000`. Le code de la tâche `B` est présenté ci-dessous.

```
let taskB () : MMU_Safe (unit)
    (requires λ c0 l0 p0 →  tpe_allocated c0.tpe_reflection 0ul)
    (ensures λ c0 x c1 l0 l1 p0 p1 →
        (tpe_allocated c1.tpe_reflection 0ul)
        ∧ (modifies_only c0 c1 [0ul]))
=
    let i0 = read A 0ul 0ul in
    let i1 = read A 0ul 1ul in
    write B (add_mod i0 i1) 0ul 0ul
```

Nous avons deux programmes qui utilisent la même adresse `0000` mais sans entrer en collision. En effet, si nous exécutions la tâche `A` puis deux fois la tâche `B`, le résultat calculé serait le même puisque l'adresse `0000` dans la tâche `A` ne sera jamais écrite par la tâche `B`. Nous faisons

remarquer que nous déclarons une variable temporaire – sur la pile de variables locales – qui n'est pas prise en charge par notre MMU. Ceci est une limitation actuelle de notre système, qui ne permet pas de gérer la mémoire complète des programmes utilisant notre MMU pour leurs variables locales et leur code.

Ainsi cet exemple montre que la mémoire de la tâche `B` contient un résultat obtenu à partir d'informations contenues dans la mémoire de la tâche `A`. Si nous empêchons ces formes d'échanges, le confinement sera alors total et ressemblera au fonctionnement exact d'un MMU. Cependant, nous souhaitons proposer une ouverture : pouvons-nous assurer que des programmes puissent échanger des informations tout en contrôlant leur propagation dans le système ?

Troisième partie

Contrôle de flux d’information

Dans cette partie, nous présentons le second aspect de ces travaux : le CFI. Cet aspect représente le cœur de ce travail de thèse et se focalise sur la manière dont les informations sont construites dans un programme. Tous les travaux décrits dans cette partie étendent LIO, une bibliothèque de CFI décrite dans l'état de l'art. Cette partie présente les trois dernières contributions :

— Une implémentation complète de LIO dans F⋆. Cette contribution implique une définition en F⋆ de la spécification de LIO. Cela permet de vérifier automatiquement les programmes de manière similaire aux travaux (Stefan, Russo, Mitchell et al. 2011) et (Vazou et al. 2014), mais en tenant compte des contraintes de l'embarqué vues en partie II. De plus, l'expressivité de F⋆ permet de prouver des théorèmes comme vus dans l'article (Parker, Vazou et Hicks 2019). Cette contribution adapte en F⋆ et dans un cadre d'informatique embarquée, l'état de l'art du CFI écrit en Haskell et LiquidHaskell. Cette contribution est illustrée par un exemple d'application.

— Une combinaison entre labellisation statique et dynamique. Cette contribution a pour objectif d'améliorer les performances en permettant une labellisation statique – elle sera enlevée à la compilation – lorsque la preuve est facile. L'idée est d'enlever les vérifications dynamiques qui peuvent être déduites par une vérification formelle.

— Une tentative d'automatisation de preuve de la non-interférence. Cette contribution consiste à donner un algorithme qui génère automatiquement un théorème et essaye de le prouver. Bien que limité par la métaprogrammation en F⋆, cette contribution met en lumière des pistes pour automatiser la preuve de propriété.

Cette partie se découpe en quatre chapitres. Le chapitre 7 donne une définition de LIO et son implémentation correspondante en F⋆. Ce chapitre donne toutes les notions essentielles pour comprendre ce qu'est LIO. Il présente les prédicats qui permettent de modéliser la propagation de l'information à chaque labellisation et chaque délabellisation.

Le chapitre 8 illustre la définition de LIO du chapitre 7. Cette illustration permet de reprendre le concept de gestionnaire de mémoire et de l'adapter au CFI. Cet exemple a pour objectif de mesurer la différence de performance entre la labellisation statique et dynamique. Cependant, il n'y a pas eu de comparaison avec le MMU puisque ce gestionnaire de mémoire est différent. En effet, il peut partager des zones de la mémoire entre différentes tâches.

Dans le chapitre 8, nous convertissons un gestionnaire mémoire utilisant de la labellisation dynamique ou statique. L'idée du chapitre 9 est de combiner la labellisation statique et dynamique au sein d'un même programme. Cette combinaison génère des défis en termes de preuves de programmes pour passer de l'un vers l'autre.

Enfin le chapitre 10 présente la preuve de non-interférence. Cette automatisation est présentée en section 10.1 puis un formalisme permet d'illustrer en section 10.2 le fonctionnement d'une telle preuve tel qu'utilisé dans les articles (Stefan, Russo, Mitchell et al. 2011) et (Parker, Vazou et Hicks 2019). Une implémentation très limitée est présentée en section 10.3.

En résumé la partie III aborde le CFI en F⋆ en gardant les contraintes liées à l'informatique

embarquée. Elle présente trois contributions allant dans le sens de l'automatisation de la preuve de programme et de l'optimisation des performances pour des microcontrôleurs.

Chapitre 7

Library Input Output

LIO est une bibliothèque de CFI qui permet d'ajouter du CFI dynamique dans des programmes écrits en Haskell. Cela donne une manière d'isoler des portions de code selon leur rôle ((Stefan, Russo, Mitchell et al. 2011), (Stefan, Russo, Mitchell et al. 2017)).

L'exemple utilisé par le papier (Stefan, Russo, Mitchell et al. 2011) est celui d'un logiciel de gestion de conférences. En effet, lors de la soumission d'un papier certaines informations sont masquées aux relecteurs, comme l'identité des auteurs. L'idée est d'utiliser le CFI pour isoler les requêtes des relecteurs concernant les informations sensibles. Pour cela, LIO va ajouter une information supplémentaire qui permet de garder une traçabilité de l'information. Le CFI va permettre d'appliquer la politique de sécurité et une preuve de non-interférence de LIO permet de montrer la justesse de cette application (Stefan, Russo, Mitchell et al. 2017). Pourquoi souhaitons-nous réécrire LIO en F⋆ ?

La réponse à cette question est simple : le manque de ressources dans l'informatique embarquée rend l'utilisation de langages de haut niveau comme Haskell très difficile. Cependant, ce langage – comme de nombreux langages fonctionnels – garantit l'absence d'effets de bord, ce qui est essentiel pour garantir le bon fonctionnement de LIO. En effet, dans un langage impératif l'altération de l'état global est normale, ce qui ne l'est pas en Haskell à moins d'utiliser des monades – cela peut être vu dans ce cas comme un paradigme qui permet de contextualiser des fonctions. La programmation par monades est plus contraignante, mais elle permet d'avoir plus de garanties sur la manière dont ce contexte est mis à jour (Ariola et Sabry 1998). Ainsi, l'implémentation de LIO en Haskell est garantie puisque son contexte CFI est protégé par le langage, ce qui n'est pas le cas des langages systèmes comme le langage C. Dans ce dernier langage, il n'est pas possible de garantir une bonne implémentation puisqu'il est impossible de contraindre l'implémentation du CFI. Notre objectif est d'utiliser le langage F⋆ pour garantir que le code validé ou extrait n'a pas d'altération de son contexte CFI.

Dans ce chapitre, nous allons présenter le fonctionnement, l'implémentation et l'utilisation d'une implémentation LIO minimale en F⋆. Nous prendrons un exemple de mémoire partagée afin d'illustrer la gestion des flux d'information entre différents acteurs. Toutefois, nous ne traiterons pas des problématiques liées à la concurrence entre ces acteurs : nous laissons les questions de parallélisme comme travaux futurs.

La bibliothèque LIO repose sur un nombre très réduit de prérequis et de fonctions. Elle utilise une liste de labels qui correspondent aux acteurs du système et qui peuvent être combinés pour former des labels de groupes. Dans la version Haskell de LIO, les labels permettent d'encoder des contraintes précises (Stefan, Russo, Mazières et al. 2011) mais dans notre travail nous utiliserons une version simplifiée. Nos labels ne peuvent exprimer qu'une conjonction de labels.

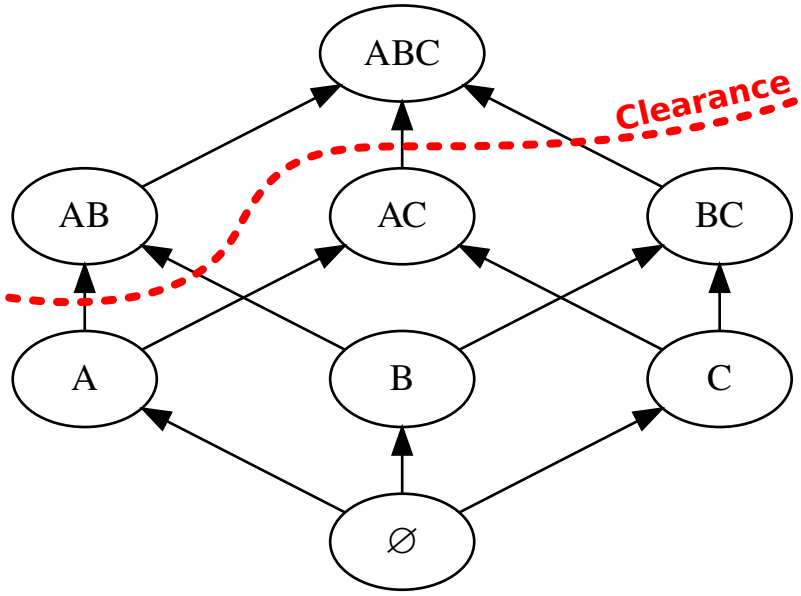

FIGURE 7.1 – Exemple de treillis pour 3 tâches A, B et C

Notre exemple comporte trois tâches : `A`, `B` et `C`. Ces petits programmes utilisent une mémoire partagée. Ce genre de mémoire permet d'écrire des mécaniques de communication entre processus : Inter-Process Communication (IPC). Ces moyens de communications sont très utilisés dans la programmation de système d'exploitation. Notre objectif est d'illustrer comment le CFI peut appliquer une politique de sécurité sur ces trois tâches.

7.1 Treillis borné

Cette structure algébrique définit un ensemble partiellement ordonné qui possède deux lois internes que nous noterons $\sqcup$ et $\sqcap$ dans notre document. L'opération $\sqcup$ permet de fusionner deux labels et l'opération $\sqcap$ permet de faire l'intersection de deux labels. Ces deux lois sont *commutatives* et *associatives* et sont reliées par la loi d'absorption :

$$\oplus \in \{\sqcup, \sqcap\}.\ \forall a\, b.\ a \oplus b = b \oplus a \tag{7.1}$$

$$\oplus \in \{\sqcup, \sqcap\}.\ \forall a\, b\, c.\ (a \oplus b) \oplus c = a \oplus (b \oplus c) \tag{7.2}$$

$$\forall a\, b.\ a \sqcup (a \sqcap b) = a = a \sqcap (a \sqcup b) \tag{7.3}$$

Nous utilisons des treillis bornés. Cela implique que nous ayons toujours un minimum. De plus, un treillis doit respecter les propriétés d'un ordre partiel : *réflexivité*, *antisymétrie* et *transitivité*. Nous vérifions ces propriétés dans les structures données par le développeur.

Classe de types En F⋆ nous pouvons définir une structure qui doit respecter une spécification donnée. Nous définissons donc un treillis générique qui permet de s'adapter à toutes les implémentations qu'un utilisateur souhaite avoir. Cela permet de rendre notre implémentation indépendante du scénario d'utilisation et de la structure des labels.

Notre treillis doit comporter trois opérations : $\sqsubseteq$, $\sqcap$ et $\sqcup$. Ces opérations servent à vérifier si deux labels sont comparables ($\sqsubseteq$), avoir l'intersection entre deux labels ($\sqcap$) et combiner deux labels ($\sqcup$). Le code ci-dessous donne la signature de ces opérations pour des labels de type `a`.

```
class lattice a = {
    ⊑:a → a → bool;
    ⊓:a → a → a;
    ⊔:a → a → a;
    bottom:(v: a {∀ l. v ⊑ l});

    (* partial order rules *)
    lawFlowReflexivity : l:a → Lemma (l ⊑ l);
    lawFlowAntisymetry : x: a → y: a →
        Lemma (ensures x ⊑ y ∧ y ⊑ x ⟹ x == y);
    lawFlowTransitivity : x: a → y: a → z: a →
        Lemma (ensures x ⊑ y ∧ y ⊑ z ⟹ x ⊑ z);

    (* lattice rules *)
    lawJoin: z: a → x: a → y: a →
        Lemma (ensures (z == x ⊔ y)
            ⟺ (x ⊑ z ∧ y ⊑ z ∧ (∀ l . x ⊑ l ∧ y ⊑ l ⟹ z ⊑ l)));

    lawMeet: z: a → x: a → y: a →
        Lemma (ensures (z == x ⊓ y)
            ⟺ (z ⊑ x ∧ z ⊑ y ∧ (∀ l . (l ⊑ x ∧ l ⊑ y) ⟹ l ⊑ z)));
}
```

L'utilisateur doit fournir également la valeur minimale (`bottom`) que le treillis peut avoir. Le raffinement de cette valeur force cette valeur à être minimale pour la relation $\sqsubseteq$. De plus, les trois lemmes `lawFlowReflexivity`, `lawFlowAntisymetry` et `lawFlowTransitivity` contraignent $\sqsubseteq$ afin que ce soit un ordre partiel.

Enfin, les deux opérations qui servent à manipuler les labels ($\sqcap$ et $\sqcup$) sont conditionnées par deux lemmes supplémentaires : `lawJoin` – la borne supérieure d'un tuple de labels sera inférieure ou égale à tout label supérieur ou égal à ce tuple – et `lawMeet` – un label inférieur à un tuple de labels sera également inférieur à leur borne inférieure. Ainsi l'utilisateur peut fournir n'importe quelle structure qui respecte toutes ces conditions et il peut fournir une preuve pour chacun des lemmes pour assurer la justesse de sa structure.

Si nous reprenons notre exemple de trois tâches, nous avons un treillis assez simple qui

correspond à tous les cas possibles d'une information qui peut n'appartenir à aucune tâche (∅), ou appartenir à une tâche (par exemple `A`), ou être partagée entre deux tâches (`AB`). Des labels pour un tel système peuvent être définis comme ci-dessous.

```
type labels = | Nothing | A | B | C | AB | BC | AC | ABC
```

Nous écrivons alors trois opérateurs – `lCanFlow`, `lJoin` et `lMeet` – qui correspondent à ceux décrits dans la définition du treillis. Les labels et ces opérateurs permettent de définir un treillis illustré par la figure 7.1. Afin de simplifier la description du code, nous ne donnerons pas le contenu des opérateurs. Cependant, nous donnons le code ci-dessous qui initialise le treillis :

```
let latticeUsed : lattice labels = {
    ⊑ = lCanFlow;
    ⊔ = lJoin;
    ⊓ = lMeet;
    bottom = Nothing;
    lawFlowReflexivity  = (λ _ → ());
    lawFlowAntisymetry  = (λ _ _ → ());
    lawFlowTransitivity = (λ _ _ _ → ());
    lawJoin = (λ _ _ _ → ());
    lawMeet = (λ _ _ _ → ());
}
```

Nous faisons remarquer que les preuves données sont vides – `λ _ → ()` est une preuve vide. Comme pour beaucoup d'exemples que nous présentons, le solveur SMT est capable de résoudre la plupart des cas triviaux, d'autant que dans cet exemple, le treillis est fini et est de petite taille. Maintenant que nous avons défini les labels, nous allons montrer comment relier ces labels avec des données.

7.2 Effet LIO

Dans notre bibliothèque de CFI, nous avons un label global variable qui indique la connaissance du programme à un instant donné. Cela permet de savoir quelles sources d'information ont été utilisées. Ainsi, l'état de connaissance d'un programme évolue dans le temps. Dans notre exemple, une tâche ayant des connaissances sur la tâche `B` qui lit une donnée labellisée `A` verra ses connaissances inclure `A` et `B` – soit `AB` dans notre treillis illustré sur la figure 7.1.

Un programme ne doit pas perdre la trace de ses connaissances. Cela est possible en assurant qu'un label courant ne peut qu'augmenter sa sensibilité avec le temps. En effet, lorsqu'un programme lit des informations secrètes, il devient de sensibilité secrète. S'il revient à une sensibilité publique – sans avoir de moyen de déclassifier l'information – cela veut dire qu'il a perdu la trace de ses connaissances : il a oublié qu'il a lu une information secrète. Si cette solution a l'air simple, elle pose le problème de contamination des labels. En effet, si un programme

connaît des informations provenant de tous les labels, il se comportera comme un programme qui n'a plus de CFI. Nous verrons plus loin comment il est possible de contourner ce problème avec une exécution temporaire définie dans (Stefan, Russo, Mitchell et al. 2011).

Autorisation Notre label courant évolue dynamiquement et le CFI a pour objectif de contrôler cette évolution. Appliquer une politique de sécurité pour contraindre l'évolution du label courant permet d'empêcher un programme de mélanger n'importe quelles informations. Imaginons que la tâche `A` soit un canal de communication vers internet et que la tâche `B` soit un gestionnaire de mots de passe. Exprimer `AB` comme autorisation maximale revient à séparer le treillis en deux. Tout label plus grand sera alors interdit.

Pour des raisons techniques, la politique de sécurité s'exprime dans cette implémentation par une fonction qui prend le label actuel et retourne vrai ou faux. En effet, nous utilisons des labels simples qui ne permettent d'exprimer que des conjonctions, contrairement à (Stefan, Russo, Mazières et al. 2011). Cela est limitant si nous voulons contraindre l'exécution à une politique donnée. Par exemple, une politique considérant que seuls sont légitimes tous les labels inférieurs ou égaux à `AC` ou `BC`. Cette politique refuse une information qui mélange des données venant de `A` et venant de `B` mais pas les autres combinaisons. Utiliser une fonction qui valide ou non le label nous permet d'exprimer une politique de sécurité plus expressive comme celle qui considère tous labels inférieurs ou égaux à `AC` ou `BC` comme seuls légitimes – `AB` et `ABC` sont alors illégitimes et la fonction `neverAB` correspond à l'autorisation de la figure 7.1. Le code `neverAB` ci-dessous montre une fonction qui retourne vrai ou faux en fonction du label courant passé en paramètre.

```
let neverAB l = l ⊑ AC || l ⊑ BC
```

Contexte Nous venons de voir qu'un contexte de CFI contient au moins deux informations : le label courant et le label maximum – ici représenté par une fonction. Ce contexte permettra de savoir à chaque instruction ce que le programme peut faire et quel est son niveau de sensibilité. Nous pouvons donner une implémentation de ce contexte – appelé `lioCTX` – ci-dessous :

```
type clearance = labels → Tot bool

noeq
type lioSTATE = {cur:labels; cle:clearance}

type lioCTX = lioSTATE
```

Ce contexte contient un label courant `cur` et une fonction `cle`. Les fonctions étant non comparables, nous précisons à F⋆ par le mot clef `noeq` que la structure ne peut pas être comparée. L'effet du CFI est implémenté de manière similaire aux effets implémentés dans la partie II. Le type `clearance` définit le type des fonctions qui retournent vrai ou faux selon que le label passé

en paramètre respecte ou non la politique de sécurité. Nous précisons que `lioCTX` et `lioSTATE` sont identiques mais nous avons deux types puisque nous ferons une distinction entre `lioCTX` qui pourra être effacé à l'extraction et `lioSTATE` qui sera utilisé par la Trusted Computing Base (TCB).

```
total new_effect LIO = S.STATE_h lioCTX
```

Nous passerons sur les détails techniques qui expliquent la connexion entre le monde `Pure` et ce nouvel effet. Cependant, ces détails sont expliqués dans la section 5.2 du chapitre 5. Le code ci-dessous permet à du code géré par LIO d'inclure du code fonctionnellement pur qui n'interagit pas avec les entrées/sorties du programme.

```
unfold let lift_pure_global (a:Type)
    (wp:pure_wp a)
    (context:lioCTX)
    (p:LIO?.post a)
=
    wp (λ a → p (a, context))
sub_effect PURE ~> LIO = lift_pure_global
```

De la même manière, la transformation d'une fonction en triplet de Hoare avec des pré- et post-conditions se définit comme vu précédemment dans le chapitre 5 et le chapitre 2. Dans le code ci-dessous, nous écrivons cette transformation pour simplifier l'écriture de nos programmes.

```
effect Lio (a:Type)
    (pre: lioCTX → Type0)
    (post: lioCTX → a → lioCTX → Type0)
=
    LIO a (λ n0 p → pre n0
        ∧ (∀ a n1.
            pre n0 ∧ post n0 a n1 ⟹ p (a, n1)))
```

Maintenant que nous avons exprimé notre effet, nous allons montrer comment le modifier et y accéder. En fait, lorsqu'une fonction de LIO a besoin de modifier l'état global de CFI du programme, elle a besoin de lire et de mettre à jour le contexte `lioSTATE`. Tout d'abord nous donnons le code de l'accesseur du contexte :

```
noextract
let getSTATE (_:unit) : Lio (lioCTX)
    (requires λ _ → ⊤)
    (ensures λ c0 x c1 → c0 == c1 ∧ x == c1)
=
    LIO?.get ()
```

Cette fonction ne prend aucun paramètre. Elle retourne une valeur de type `lioSTATE` dont la valeur est celle du contexte courant (`x == c1`) et elle ne modifie absolument pas le contexte (`c0 == c1`). Cependant, nous faisons remarquer que cette fonction ne doit pas être utilisée en dehors des primitives de LIO. Le code de cette fonction sera remplacé par une fonction écrite en langage C qui sera adaptée à la plateforme ciblée. Le mot-clef `noextract` indique à F⋆ de ne pas extraire le corps de la fonction. Nous soulignons que nous utilisons `LIO?.get` encapsulé dans une fonction. En effet, F⋆ génère des accesseurs et mutateurs pour les effets, mais ils ne sont pas extractibles. En utilisant une fonction, cela nous permet d'éventuellement avoir une forme extractible et de les spécifier plus précisément.

Pour la mise à jour, nous devons respecter certains invariants comme l'augmentation inéluctable du label courant (`c0.cur ⊑ c1.cur`) et la réduction de la permissivité d'un programme (`c1.cle l ⟹ c0.cle l`). Nous appelons cette propriété la monotonie du contexte. Ainsi, le programme ne peut que connaître davantage de labels et avoir une politique de sécurité de plus en plus fine avec le temps.

```
let monotonicity (c0 c1:lioCTX) : GTot (Type) =
    (∀ (l:labels). c1.cle l ⟹ c0.cle l
        ∧ c0.cur ⊑ c1.cur)
```

Dans le mutateur de notre effet, le développeur pourra fournir n'importe quel contexte afin de faire évoluer l'état du CFI. La postcondition indique clairement que le nouveau contexte est équivalent au contexte `s` donné en paramètre. Le code du mutateur est donné ci-dessous :

```
noextract
let setSTATE_unsafe (s:lioCTX) : Lio (unit)
    (requires λ c0 → ⊤)
    (ensures λ c0 x c1 → s == c1)
=
    LIO?.put (s)
```

Cependant, ce mutateur est dangereux puisqu'il permet de mettre à jour n'importe quel contexte. En effet, nous pourrions passer d'un contexte sensible à un contexte public, ce qui dans la plupart des cas est très dangereux. Par exemple, un utilisateur peut lire une variable secrète puis se déclarer de sensibilité publique pour écrire dans un canal public, ce qui représente une fuite d'information.

Nous verrons plus loin un cas où nous sommes obligés d'utiliser ce mutateur. Cependant, dans tous les autres cas nous utilisons un mutateur qui exige que le nouveau contexte soit monotone par rapport à l'ancien. Il est assez facile d'écrire ce mutateur en appelant le mutateur dangereux uniquement lorsque la monotonie du contexte est respectée. Le code est donné ci-dessous :

```
let setSTATE (s:lioCTX) : Lio (unit)
    (requires λ c0 → monotonicity c0 s)
```

```
    (ensures λ c0 x c1 → s == c1)
=
    setSTATE_unsafe s
```

Le mieux est de définir les accesseurs et mutateurs directement dans la définition de l'effet mais pour des raisons techniques nous exposons ces fonctions. Les pré- et post-conditions protègent la manière dont ces fonctions sont utilisées. C'est pourquoi `setSTATE_unsafe` et `getSTATE` ne s'extraient pas en langage C : ils sont remplacés par un code de la TCB déjà implémenté en C. Cette approche permet également de pouvoir extraire les appels de ces fonctions afin de fournir une implémentation en C. L'intérêt d'utiliser la vérification est de réduire la TCB qui, dans cet exemple, est réduite aux fonctions `getSTATE` et `setSTATE`.

Par exemple, la TCB de LIO en langage C sous Linux est très courte. Dans le code suivant, nous donnons un exemple d'implémentation système que nous utilisons dans notre exemple. Bien que dépendant du code F⋆ qui définit le type `lioSTATE`, le code contient la définition des deux fonctions que nous avons présentées précédemment, `getSTATE` et `setSTATE_unsafe` :

```
static lioSTATE glob;

lioSTATE getSTATE(void *n){
    return glob;
}

void setSTATE_unsafe(lioSTATE g){
    glob = g;
}
```

Nous précisons que `lioCTX` et `lioSTATE` sont strictement équivalents. Cependant, nous montrons comment effacer le contexte CFI plus loin dans ce chapitre. Cela rend inutiles les appels à `getSTATE` et à `setSTATE_unsafe` et cela explique pourquoi dans ce cas, il peut y avoir une divergence entre `lioCTX` et `lioSTATE`.

Cependant, en utilisant pleinement les possibilités de F⋆ et L⋆, nous pourrions enlever toute dépendance non vérifiée dans notre code, mais cela rendrait notre exemple très dépendant de L⋆. Dans ce cas, une implémentation de LIO telle que présentée complexifierait le code. Nous présentons les résultats d'un tel travail dans la section 9. Maintenant que nous avons un moyen de tracer l'évolution de la connaissance d'un programme nous pouvons implémenter les primitives de LIO.

7.3 Valeur labellisée

Pour attacher une valeur et un label, nous définissons une structure que nous appelons valeur labellisée. Cette structure contient deux champs : *data* qui contient la valeur à protéger et *tag*

qui contient la labellisation de cette valeur. Une telle structure peut être définie de la manière suivante :

```
type labeled (a:Type) = {
    value:a;
    tag:labels
}
```

Cette structure sera utilisée pour le reste de l'implémentation. Elle est polymorphique – c'est-à-dire définie pour n'importe quel type (`Type`). Ainsi, il est possible d'avoir par exemple des `labeled int`. La structure comme ses accesseurs et mutateurs sera déclinée pour tous les types de données que nous passerons en paramètre. Nous faisons remarquer que nous notons parfois une valeur labellisée $\langle$`value,tag`$\rangle$ dans notre document afin de montrer la valeur et le label qu'elle contient.

Label Parfois il est intéressant d'obtenir le label d'une valeur labellisée. Pour cela il suffit d'exprimer une fonction qui retourne le label. Cette fonction – appelée `labelOf` – n'a pas besoin d'être spécifiée du point de vue du CFI puisqu'elle ne fuite pas la valeur mais juste son niveau de sécurité. En effet, cette fonction appliquée à la valeur labellisée $\langle$`5,A`$\rangle$ ne retournera que `A`.

```
let _labelOf (#a:Type) (vl:labeled a) : Ghost (labels)
    (requires ⊤)
    (ensures λ x → x == vl.tag)
=
    vl.tag
```

Cette fonction peut être très utile pour les prédicats et nous pouvons être tentés de l'utiliser dans un programme pour décider en fonction des labels attachés aux informations de ne pas augmenter le label courant. Toutefois, cette approche est dangereuse puisqu'elle permet de faire fuir une information en fonction du label de la valeur. La solution consiste à augmenter le label courant en fonction du label de la valeur labellisée. Une telle fonction s'exprime de la manière suivante :

```
let labelOf (#a:Type) (vl:labeled a) : Lio (labels)
    (requires λ c0 → c0.cle (c0.cur ⊔ _labelOf vl))
    (ensures λ c0 x c1 → c1 == {cur=(c0.cur ⊔ _labelOf vl); cle=c0.cle}
        ∧ monotonicity c0 c1
        ∧ x = vl.tag)
=
    let c0 = getSTATE () in
    setSTATE ({cur=(c0.cur ⊔ vl.tag); cle=c0.cle});
    vl.tag
```

Cette fonction `labelOf` impose que le contexte LIO soit autorisé à unir le label courant au label de la valeur labellisée. Puis elle assure dans la post-condition que le label courant augmente

`(c0.cur ⊔ _labelOf vl)`, que le contexte LIO respecte la monotonie et que le retour de la fonction est identique au label de la valeur labellisée. De fait, cela implique que l'accès au label courant soit protégé ou inaccessible. Dans ce travail, nous supposerons que la fonction d'accès au contexte est inaccessible.

Valeur Lors de l'écriture des prédicats, il est pratique d'inspecter la valeur d'une valeur labellisée. Cette fonction – appelée `valueOf` – ne doit pas être utilisée dans le code d'un programme puisqu'elle n'a pas de sens à l'exécution. En effet, l'inspection n'a pas d'impact sur le contexte global. C'est pour cette raison que nous déclarons cette fonction dans l'effet `GTot` qui est effacé.

```
let valueOf (#a:Type) (vl:labeled a) : Ghost a
    (requires ⊤)
    (ensures λ x → x == vl.value)
=
    vl.value
```

De même que `_labelOf`, cette fonction n'ayant pas d'impact sur le CFI ne nécessite pas de spécification précise. C'est pour cela que nous relions juste le retour de la fonction avec le champ `value` de la structure.

Maintenant que nous avons un contexte et des valeurs labellisées, nous pouvons écrire les primitives qui servent à manipuler les valeurs labellisées en tenant compte du contexte.

7.4 Labellisation

Cette étape consiste à attacher un label à une donnée. La valeur du label importe peu du moment que le label choisi est supérieur au label courant. Par exemple, si nous sommes dans un environnement de sensibilité publique, nous pouvons dire que la valeur `1` est de niveau secret. Cependant, si nous sommes dans un niveau de sensibilité secret, nous ne pouvons pas dire que la valeur `1` est de niveau public, car cette constante peut dépendre d'une information secrète. C'est pourquoi notre fonction `label` impose que le label fourni soit supérieur ou égal au label courant (`c0.cur ⊑ l`). Pour une meilleure lisibilité, la pré- et la post-condition sont notées dans des prédicats à part afin d'être réutilisées dans le reste du document.

```
unfold let label_pre c0 l = c0.cur ⊑ l ∧ c0.cle l
unfold let label_post c0 x c1 v l = x == {value=v; tag=l} ∧ c0 == c1

let label (#a:Type) (v:a) (l:labels) : Lio (labeled a)
    (requires λ c0 → label_pre c0 l)
    (ensures λ c0 x c1 → label_post c0 x c1 v l)
=
    {value=v; tag=l}
```

Cette simple fonction permet de construire la structure labellisante. Cette structure n'est pas disponible à l'extérieur du module de LIO et le développeur ne peut créer des labels qu'uniquement via `label`. Nous attirons l'attention sur le fait que labelliser une valeur ne change pas le contexte du CFI (`c0 == c1`).

7.5 Délabellisation

Une fois une valeur labellisée, il n'est pas possible de la retrouver sans la délabelliser. Cela peut se faire grâce à une fonction – appelée `unlabel` qui retourne la valeur d'une valeur labellisée en mettant à jour le contexte du CFI. La précondition est toujours vraie puisqu'il est possible de délabelliser toute valeur grâce à la propriété de l'opérateur ⊔ : soit le label courant est inférieur et il sera mis à jour, soit il est supérieur et ne sera pas mis à jour.

```
unfold let unlabel_pre c0 vl = c0.cle (c0.cur ⊔ _labelOf vl)
unfold let unlabel_post c0 x c1 vl =  x == vl.value
    ∧ c1 == {cur=(c0.cur ⊔ _labelOf vl); cle=c0.cle}
    ∧ monotonicity c0 c1

let unlabel (#a:Type) (vl:labeled a)  : Lio (a)
    (requires λ c0 → unlabel_pre c0 vl)
    (ensures λ c0 x c1 → unlabel_post c0 x c1 vl)
=
    let c0 = getSTATE () in
    setSTATE ({cur=(c0.cur ⊔ _labelOf vl); cle=c0.cle});
    vl.value
```

Cette fonction assure que la modification du contexte CFI reste monotone grâce au prédicat `monotonicity`. Elle met à jour le contexte en augmentant le label courant (`c0.cur`) avec le label de la valeur labellisée. Nous ajoutons qu'accéder à la valeur labellisée est très simple (`vl.value`) et c'est pour cette raison que les primitives protègent cette structure afin de forcer une utilisation correcte du CFI.

7.6 Exécution labellisée

Nous avons maintenant une bibliothèque qui permet de gérer un CFI. Cependant, nous avons un problème de contamination des labels. Pour illustrer ce problème, imaginons la fonction suivante exécutée avec une autorisation totale `cle == (λ _ →true)` :

```
let cle (l:labels) : bool = true

let label_creep #a (vl:labeled a * labeled a * labeled a) : Lio (unit)
    (requires λ c0 → let (vl1,vl2,vl3) = vl in
```

```
        c0.cle == cle
        ∧ c0.cur = Nothing
        ∧ _labelOf vl1 = A
        ∧ _labelOf vl2 = B
        ∧ _labelOf vl3 = C)
    (ensures λ c0 x c1 → c1.cur = ABC ∧ c0.cle == c1.cle)
=
    let (vl1,vl2,vl3) = vl in
    let v1 = unlabel vl1 in
    let v2 = unlabel vl2 in
    let v3 = unlabel vl3 in
    ()
```

Si nous utilisons cette fonction dans n'importe quel programme, le reste de l'exécution sera retenu dans le niveau de sécurité maximum (`ABC`). Il ne sera plus possible d'écrire dans des entrées/sorties de sensibilités inférieures. C'est pourquoi il existe une fonction – appelée `toLabeled` – qui exécute une fonction sous un niveau d'exécution temporaire puis restaure l'ancien niveau. Dans notre exemple, cette fonction permettrait d'exécuter la fonction `label_creep` puis de retourner la valeur labellisée ⟨`()`,`ABC`⟩. Nous pouvons alors appeler le programme `label_creep` de la manière suivante :

```
let no_label_creep #a (vl1 vl2 vl3:labeled a) : Lio (unit)
    (requires λ c0 → c0.cle == cle
        ∧ cle ABC
        ∧ c0.cur = Nothing
        ∧ _labelOf vl1 = A
        ∧ _labelOf vl2 = B
        ∧ _labelOf vl3 = C)
    (ensures λ c0 x c1 → c1.cur = Nothing ∧ c0.cle == c1.cle)
=
    let v = toLabeled ABC cle (label_creep) (vl1,vl2,vl3) in
    ()
```

Nous faisons remarquer que le label courant est identique à celui avant l'appel de `toLabeled`. La figure 7.2 illustre le comportement de `toLabeled`. En effet, dans la fonction `no_label_creep` l'appel à la fonction `label_creep` via `toLabeled` permet d'exécuter cette dernière avec un contexte LIO temporaire et de retourner une valeur labellisée au niveau de ce contexte temporaire.

Cette nouvelle primitive de LIO a pour paramètre une fonction et son paramètre, un label maximal et une fonction qui détermine la politique de sécurité – l'autorisation – de cette exécution. Avant de décrire le code de `toLabeled`, nous allons définir les pré- et post-conditions.

```
let toLabeled_pre pre post c0 l (c:clearance) params =
```

```
    label_pre c0 l
    ∧ pre params c0
    ∧ (∀ x c1. post params c0 x c1 ⟹
        (monotonicity c0 c1
        ∧ c l
        ∧ c1.cur = l
        ∧ c1.cle == c))

let toLabeled_post post c0 x c1 l c params =
    c0.cur = c1.cur
    ∧ c1.cle == c
    ∧ post params c0 (valueOf x) ({cur=l; cle=c})
    ∧ monotonicity c0 c1
```

La précondition doit valider la labellisation au label donné `l`, la précondition (`pre`) de la fonction que `toLabeled` exécutera et sa postcondition (`post`). Si cette dernière est valide, elle doit assurer certaines propriétés : monotonie entre contexte initial et final (le prédicat `monotonicity c0 c1`), respect de l'autorisation (`c l`) et un contexte contraint – label courant identique au contexte initial et autorisation identique à celle passée en paramètre : (`c1.cur = l ∧ c1.cle == c`). La postcondition est plus simple puisqu'elle valide uniquement les propriétés assurées dans la précondition (monotonie,…) ainsi que la postcondition (`post`) de la fonction que `toLabeled` exécutera. Nous faisons remarquer deux points importants :

— Dans les pré- et post-conditions de `toLabeled` nous ne parlons que des pré- et post-conditions de la fonction (`pre`, `post`) que nous appellerons. Cela est dû à la généricité de `toLabeled` qui est utilisable pour toute fonction.

— Nous avons pris la liberté de contraindre l'état du CFI final. En effet, nous avons souhaité montrer une implémentation minimale et avoir un contexte CFI plus générique aurait augmenté considérablement l'effort de preuve.

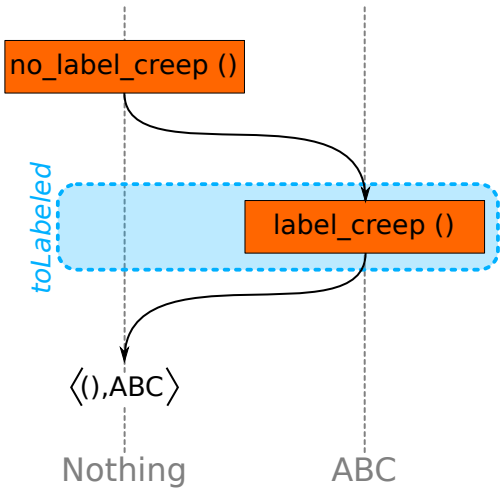

FIGURE 7.2 – Illustration de `toLabeled` dans la fonction `no_label_creep`

Ainsi, notre fonction d'exécution labellisée sauvegarde le contexte avant (`n0`) et après (`n1`) l'appel de la fonction – appelée `call`. Le résultat de cette fonction est stocké dans la variable `r`. À ce stade la valeur n'est pas protégée. La création du contexte final (`n2`) contient le label courant du contexte initial (`n0`) et l'autorisation du contexte après l'appel de `call`. C'est ici que notre contrainte est visible : au lieu de demander une autorisation au moins aussi restrictive nous demandons une autorisation identique. Avant de mettre à jour le contexte, nous labellisons le résultat de `call`. Cette ultime labellisation doit être faite dans le contexte CFI de la fonction `call` qui est plus sensible que le contexte initial (`n0`).

```
let toLabeled   #a
                #b
                (#pre)
                (#post)
                (l:labels)
                (c:clearance)
                ($call:((v:b) → Lio a (pre v) (post v)))
                (params:b)
        : Lio (labeled a)
    (requires λ c0 → toLabeled_pre pre post c0 l c params)
    (ensures λ c0 x c1 → toLabeled_post post c0 x c1 l c params)
=
    let n0 = getSTATE () in                 (* récupération du contexte *)
    let r  = (call params) in               (* appel de la fonction *)
    let n1 = getSTATE () in                 (* récupération du contexte *)
    let n2 = {cur=n0.cur; cle=n1.cle} in    (* création du contexte final *)
    assert(post params n0 r ({cur=l; cle=c}));
    let vl = label r l in                   (* labellisation du résultat *)
    setSTATE_unsafe (n2);                   (* sauvegarde contexte final *)
    vl                                      (* valeur retournée *)
```

Nous soulignons que nous utilisons `setSTATE_unsafe` pour mettre à jour le contexte. Dans ce cas précis, c'est normal puisque la monotonie n'est pas respectée dans cette primitive de LIO – `monotony n0 n2` est valide alors que `monotony n1 n2` ne l'est pas.

Nous utilisons également une subtilité du langage F⋆ pour la fonction `call`. En effet, nous utilisons l'opérateur `$` qui permet de "copier/coller" la précondition la plus faible – c'est-à-dire les pré- et post-conditions – de la fonction `call` afin que la précondition la plus faible de l'appel à `toLabeled` la prenne aussi en compte.

Dans ce chapitre, nous venons de présenter une implémentation minimale de LIO. Cette implémentation a pour objectif d'illustrer les primitives et leur fonctionnement dans cette bibliothèque mais aussi comment nous les utilisons dans nos travaux. De plus, nous avons implémenté une version vérifiée de LIO qui reprend toute la spécification présentée dans l'état de l'art ((Stefan, Russo, Mitchell et al. 2011), (Stefan, Russo, Mitchell et al. 2017)) et permet d'adapter

la vérification à toute implémentation de labels respectant un treillis générique.

Utilisation concrète de LIO

Pour illustrer cette implémentation de LIO, nous allons décrire un gestionnaire de mémoire vérifié en F⋆. L'objectif de ce programme est de partager la mémoire entre différentes tâches – dans notre cas elles seront représentées par des fonctions – tout en assurant que certaines tâches seront isolées des autres. Notre implémentation ressemble beaucoup à la modélisation de l'Arduino décrite en partie II et surtout fait écho à l'exemple décrit en section 6.2.

8.1 Description de l'application

Notre exemple d'application est une mémoire partagée. C'est-à-dire un tableau d'octets qui sont utilisés par plusieurs acteurs qui, dans notre exemple d'implémentation, sont des fonctions. Cette approche permet de gérer de la mémoire dynamique pour des systèmes embarqués ainsi que des IPC. Dans ces deux scénarios d'utilisation, un tel outil permet d'améliorer les fonctionnalités d'un système tout en garantissant certaines propriétés. Nous en vérifierons deux : 1/ la mémoire est accédée en respectant ses limites et 2/ les tâches sont isolées en utilisant LIO.

Nous structurons la mémoire comme dans le modèle mémoire du langage C. Nous utilisons des adresses qui pointent à l'intérieur de notre mémoire sur le début d'une donnée. La mémoire peut être vue comme un ruban rempli de cases dont l'adresse correspond à la position d'une case. Une case contient un octet dont les valeurs varient de 0 à 255. La figure 8.1 illustre une telle mémoire.

L'accès à la mémoire consiste à retourner la valeur de l'octet à une adresse donnée. De même l'écriture consiste à remplacer la valeur contenue à une adresse donnée avec une nouvelle valeur donnée. Ces accès doivent être bornés entre 0 et la taille de la mémoire. Dans la figure 8.1, la mémoire est bornée à 7 éléments qui sont adressés de 0 à 6.

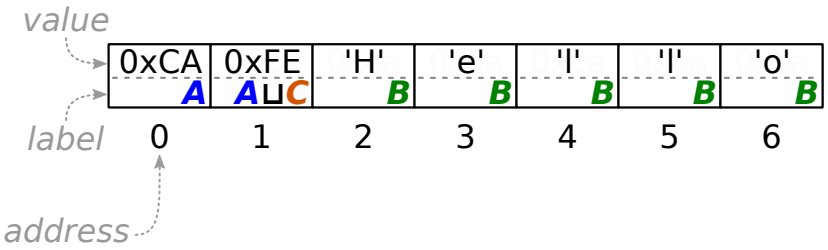

Figure 8.1 – Représentation du modèle mémoire

Si nous avons trois tâches comme dans le treillis précédent, chacune d'entre elles pourra

accéder à toute la mémoire de notre application et elles pourront ainsi échanger des informations entre elles. Dans la figure 8.1, nous ajoutons également les labels attachés à chaque cellule mémoire, qui donnent une idée des échanges entre les différentes tâches.

La figure 8.1 montre l'état d'une mémoire après une exécution. Voici un exemple d'exécution possible : la tâche `A` a pu écrire `0xcad0` à l'adresse 0 et `0xdead` à l'adresse 2, la tâche `C` a pu lire une information de la tâche `A` (disons qu'elle a lu `0xd0` à l'adresse 1) puis elle a écrit la valeur `0xfe` à l'adresse 1. Cette nouvelle valeur est dépendante de deux tâches, `A` et `C`. Enfin, la tâche `B` écrit sans rien lire de la mémoire la chaîne de caractères `"Hello"` à l'adresse 2. À ce stade de l'exécution, l'état de la mémoire sera identique à celui de la figure 8.1.

8.2 Implémentation en F⋆

Dans cette section, nous décrivons les détails techniques du gestionnaire mémoire. Cela a pour objectif de montrer comment LIO est utilisable dans un cas simple et concret.

8.2.1 Mémoire

Notre gestionnaire utilise un modèle mémoire inspiré du langage C. Cette approche nécessite de manipuler des octets qui peuvent être vus comme des entiers non signés sur 8 bits. Ainsi, une mémoire labellisée sera constituée d'octets labellisés – chaque octet aura un label. Le code ci-dessous présente leur définition en F⋆.

```
type byte = U8.t
type cell = labeled byte
```

Une fois les valeurs définies, nous définissons les adresses. Nous utilisons deux types : les adresses réelles (`ref`) et leur réflexion dans le monde de la vérification (`ref'`). Ainsi, le premier type est utilisé par le code du gestionnaire – il est borné – alors que le second type est utilisé uniquement dans le cadre de la vérification – il n'est pas borné pour des raisons techniques liées au développement de F⋆. Nous précisons que `ref'` et `memsize` – taille de la mémoire gérée – ne sont pas extraits dans le programme final :

```
noextract type ref' = int
noextract let memsize = 1024
type ref = x:U32.t{U32.v x < memsize}
```

Ainsi, le type des adresses – appelé `ref` – correspond à des adresses 32 bits que nous manipulons comme des entiers non signés (`U32.t`). En langage C, ce type de donnée peut être exprimé en `uint32_t`. Dans notre cas, les adresses sont limitées par la taille de la mémoire gérée – elles doivent être inférieures à `memsize`.

```
noextract let mkRef' (x:ref) : (r:ref'{r < memsize ∧ r ≥ 0}) = U32.v x
```

Nous notons que nous avons une fonction (`mkRef'`) pour traduire une adresse – exprimée en nombre machine – vers des nombres utilisés pour la vérification. Le type réfléchi `ref'` permet de simplifier les préconditions les plus faibles et facilite la vérification du côté de F⋆. Nous définissons également une mémoire abstraite. Ainsi, nous pouvons connaître son état à chaque moment de la vérification du programme. Comme le type `ref'`, le type `memory` n'a aucun sens à l'exécution.

```
noextract type memory = (ref' → option cell)
```

Nous soulignons la ressemblance de la modélisation à ce stade avec celle de l'Arduino dans le chapitre 5. Tout comme la modélisation de l'état des pattes, nous modélisons l'état des cellules de la mémoire par une suite infinie de lambda. Par exemple, une mémoire vierge aura la valeur `(λ r →None)` – `None` indique qu'il n'y a pas d'adresse à évaluer dans la mémoire – et une mémoire ayant une seule valeur labellisée `⟨5,A⟩` à l'adresse 11 aura la valeur `(λ r →if r = 11 then ⟨5,A⟩ else None)`.

Toutefois, les adresses ne sont pas bornées. Pour assurer que les accesseurs et mutateurs soient corrects, nous écrivons deux prédicats : l'un pour vérifier si une adresse est correcte (`bounded`), l'autre pour vérifier une mémoire (`correctMemory`).

```
let bounded (r:ref') = r ≥ 0 ∧ r < memsize
let correctMemory (m:memory) =
    ∀ (k:ref'{k ≥ memsize ∨ k < 0}). None? (m k)
```

Tout cela nous permet d'écrire les accesseurs et mutateurs pour cette modélisation de la mémoire. Nous définissons l'accesseur `get` comme une évaluation du lambda. Si nous l'exécutons sur la mémoire donnée en exemple dans le paragraphe précédent, nous obtenons `Some ⟨5,A⟩` pour `get (λ r →if r = 11 then ⟨5,A⟩ else None) 11` et `None` pour tout adresse différente de 11. `None` indique ainsi que l'adresse n'a jamais été écrite.

```
let get (m:memory) (r:ref') : Pure (option cell)
    (requires correctMemory m ∧ bounded r)
    (ensures λ x → ⊤)
= m r
```

Pour comparer deux mémoires, nous écrivons un prédicat qui retourne vrai si deux mémoires sont identiques. Ce prédicat est très similaire à celui qui compare l'état des pattes de l'Arduino. Nous donnons un argument qui exclut certaines adresses dans la comparaison. Cela nous permet d'exprimer le changement partiel de la mémoire – c'est-à-dire qu'à part un ensemble donné d'adresses la mémoire est inchangée.

```
let modifies (m0 m1:memory) (l:list ref') =
    (∀ (k:ref'{¬(L.contains k l)}). m0 k == m1 k)
```

Ce changement partiel est illustré dans la fonction `set` qui, pour une mémoire donnée, une adresse et une valeur, fabrique une nouvelle mémoire. Cette fonction a une précondition et une

postcondition qui assurent que la valeur à l'adresse donnée est correcte et que seule cette adresse est modifiée entre la mémoire donnée et la nouvelle mémoire. Le code ci-dessous met à jour une valeur dans une mémoire. Nous faisons remarquer la construction inductive de la nouvelle mémoire.

```
let set (m:memory) (r:ref') (v:option cell) : Pure (memory)
    (requires correctMemory m ∧ bounded r)
    (ensures λ x →
        correctMemory x
        ∧ ((get x r) == v)
        ∧ (modifies m x [r]))
=
    let x = (λ c → if c = r then v else m c) in
    x
```

Nous soulignons que nous utilisons des valeurs labellisées comme données stockées en mémoire. Pour simplifier la vérification, nous écrivons une fonction qui permet d'inspecter le label d'un octet en mémoire. Toutefois cette fonction retourne au moins le label `Nothing` pour n'importe quelle adresse. Cela signifie qu'un programme qui inspecte le label d'une adresse non initialisée (jamais écrite par le programme) est au niveau de sécurité le plus bas.

```
let labelOfCell (m:memory{correctMemory m}) (r:ref'{bounded r}) : GTot (labels)
    =
    let c = (get m r) in
    match c with
        | Some v → _labelOf v
        | None → Nothing
```

Nous avons maintenant la modélisation complète de la mémoire. Ce que nous montrons dans le paragraphe suivant est l'association entre cette modélisation mémoire et un état de CFI.

8.2.2 Contexte et CFI

Pour avoir une trace de notre mémoire et LIO en même temps, nous définissons un effet qui contient les deux informations. Ce nouveau contexte – appelé `memorySTATE` – doit à la fois connaître l'état de la mémoire et du CFI. Contrairement à LIO, nous ne souhaitons pas extraire la totalité de ce contexte car l'état de la mémoire n'a pas de sens au niveau de l'implémentation. De fait, nous effaçons – via le module `G.erased` du module `Ghost` – le champ `mem` qui contient notre modélisation de la mémoire.

```
noeq
type memorySTATE = {ifc:lioCTX; mem:G.erased memory}

total new_effect MEMORY = S.STATE_h memorySTATE
```

Exactement comme pour LIO, nous créons un nouvel effet. Toutefois, nous n'allons pas relier cet effet à l'effet `Pure` mais nous le relions à l'effet `LIO`. Cela permet d'avoir des fonctions sous l'effet `LIO` mais aussi sous l'effet `Pure`. La fonction d'élévation – appelée `lift_state_mem` – va indiquer comment relier le contexte du CFI de l'effet `LIO` avec la structure `memorySTATE` de l'effet `MEMORY`. Nous relions ainsi les deux effets `LIO` et `MEMORY`.

```
unfold let lift_state_mem (a:Type) (wp:LIO?.wp a) : MEMORY?.wp a
=
    λ c (p:MEMORY?.post a)
        → wp
            c.ifc
            (λ i → let v,c1 = i in
                p (v,  ({ifc=c1; mem=c.mem}))
            )
sub_effect LIO ⇝ MEMORY = lift_state_mem
```

Nous définissons également un effet `Memory` qui prend une précondition et une postcondition. Cette définition étant extrêmement similaire à celle de `Lio`, nous ne donnons pas le code.

8.2.3 Primitives

Pour rendre notre gestionnaire de mémoire utilisable, nous lui ajoutons une opération de lecture et une opération d'écriture. Dans notre cas, nous n'utilisons pas L⋆ pour la vérification de la mémoire et donc nous devons utiliser nos primitives – `read_raw` et `write_raw`. Ces dernières sont supposées correctes – elles appartiennent à la TCB – ce qui implique qu'elles doivent être vérifiées manuellement ainsi que leur spécification.

TCB

Puisque nous n'utilisons pas L⋆, nous avons un code C qui permet de stocker les valeurs labellisées en mémoire. À noter que le type `cell` est défini en F⋆ puis traduit en C. Cela permet de n'avoir que les entrées/sorties de la mémoire écrites en C. Dans certains cas – comme l'embarqué – ces entrées/sorties peuvent être différentes et sont donc séparées pour des raisons pratiques. La mémoire partagée en question est un simple tableau du type `cell` tel qu'il est extrait. Le code ci-dessous montre la déclaration de la mémoire, qui est hors de portée de F⋆.

```
static cell hwmem[0x400]; // 1024 en décimal
```

Ainsi la lecture de la mémoire consiste à lire le contenu de ce tableau où l'adresse dans notre mémoire correspond à l'index dans le tableau. Le code ci-dessous montre la fonction permettant de lire une valeur.

```
cell read_raw(uint32_t addr){
    return hwmem[addr];
}
```

De manière similaire, l'écriture de la mémoire permet d'écraser une valeur du tableau. Le code ci-dessous montre comment cette fonction est implémentée :

```
void write_raw(uint32_t addr, cell vl){
    hwmem[addr] = vl;
}
```

Nous faisons remarquer que ces trois morceaux de code sont très courts et sans aucune sécurité. En effet, F⋆ permettra de vérifier qu'ils sont toujours correctement appelés et leur relecture est facile grâce à leur petitesse. En effet, pour vérifier que ce code correspond à la spécification en F⋆, la relecture est suffisante. Cependant, modéliser ce code avec L⋆ permettrait de le garantir formellement, mais soulèverait des problèmes techniques. Nous venons de donner la TCB de notre gestionnaire de mémoire. Il s'agit d'un talon d'Achille puisque si ces fonctions sont mal écrites ou que leur spécification est mal rédigée alors tous les invariants de notre programme peuvent être faux. L'avantage est de pouvoir utiliser de la mémoire externe sur une Arduino puisque l'implémentation n'aura alors rien à voir avec cette implémentation pour Linux.

Avant de passer au reste du gestionnaire de mémoire, nous allons décrire la spécification des entrées/sorties puisqu'elles sont utilisées en F⋆ et que nous souhaitons assurer leur bonne utilisation.

Dans le cas de `read_raw`, nous devons nous assurer que la mémoire est correcte (grâce à `correctMemory c0.mem`) et que l'emplacement que nous lisons est cohérent (`Some? (get c0.mem (mkRef' r))`) : il a au moins été écrit une fois). Cette dernière contrainte est forte puisqu'elle force le développeur à prouver que les lectures ne se font que sur des emplacements initialisés.

```
assume
val read_raw : (r:ref{bounded (mkRef' r)}) → Memory (cell)
    (requires λ c0 → correctMemory c0.mem
        ∧ Some? (get c0.mem (mkRef' r)))
    (ensures λ  c0 x c1 → correctMemory c1.mem
        ∧ Some? (get c1.mem (mkRef' r))
        ∧ (let Some lv = (get c1.mem (mkRef' r)) in
            x == lv)
        ∧ c0 == c1)
```

La postcondition indique que le retour de la fonction `read` doit correspondre avec la valeur qui est dans la mémoire – c'est-à-dire que le programme a déjà écrit à cet endroit et que cette valeur est connue. Cela explique pourquoi nous indiquons que la valeur en mémoire est toujours

initialisée (`Some? (get c1.mem (mkRef' r))`) et nous montrons que cette valeur correspond au retour (`let Some lv = ... in x == lv`). De plus, le contexte LIO est inchangé (`c0 == c1`).

À l'opposé, la fonction `write_raw` aura un impact sur la mémoire mais pas sur le contexte CFI. Ainsi, nous indiquons comment la mémoire est modifiée uniquement sur l'adresse `r` (grâce à `modifies c0.mem c1.mem [(mkRef' r)]`). Comme dans la fonction `read_raw`, nous indiquons l'état de la mémoire à l'adresse `r`.

```
assume
val write_raw : (r:ref{bounded (mkRef' r)}) → (vl:cell) → Memory (unit)
    (requires λ c0 → correctMemory c0.mem)
    (ensures λ c0 x c1 → modifies c0.mem c1.mem [(mkRef' r)]
        ∧ correctMemory c0.mem
        ∧ Some? (get c1.mem (mkRef' r))
        ∧ (let Some c = (get c1.mem (mkRef' r)) in
            vl == c)
        ∧ c0.ifc == c1.ifc)
```

Maintenant, nous pouvons utiliser ces fonctions pour altérer et consulter la mémoire. Cependant, ces fonctions ne manipulent jamais les valeurs labellisées – elles ne modifient jamais le contexte CFI. En effet, ces fonctions sont indépendantes du CFI, mais la spécification indique comment les interconnecter. De cette manière, le CFI reste dans la partie vérifiée du programme, ce qui veut dire que l'utilisation du CFI est vérifiée et seules les entrées/sorties sont admises comme correctes. Nous présentons la manière d'implémenter cette partie CFI en utilisant LIO.

Lecture de la mémoire

Notre mémoire étant labellisée, lire dans la mémoire va changer notre contexte. En effet, si un programme lit une information secrète dans la mémoire il doit augmenter son niveau de sensibilité à secret. Cela ressemble à une délabellisation. Pour lire dans une mémoire, nous récupérons la valeur labellisée grâce à la primitive `read_raw` et nous la délabellisons.

```
let read (r:ref{bounded (mkRef' r)}) : Memory (byte)
    (requires λ c0 → correctMemory c0.mem
        ∧ Some? (get c0.mem (mkRef' r))
        ∧ (let Some lv = (get c0.mem (mkRef' r)) in
            unlabel_pre c0.ifc lv))
    (ensures λ c0 x c1 → correctMemory c0.mem
        ∧ Some? (get c0.mem (mkRef' r))
        ∧ (let Some lv = (get c0.mem (mkRef' r)) in
            unlabel_post c0.ifc x c1.ifc lv)
        ∧ c0.mem == c1.mem)
=
    let lv = read_raw r in
```

```
    unlabel lv
```

Ainsi, cette fonction va vérifier que nous sommes autorisés à lire l'information stockée dans la mémoire puis retourner la valeur contenue à l'adresse donnée. Contrairement à `read_raw` qui ne modifie ni le contexte CFI ni la mémoire, notre fonction `read` modifiera le contexte CFI. Notons que les pré- et post-conditions contiennent les pré- et post-conditions de la fonction `label`.

Écriture de la mémoire

De manière opposée à la lecture, l'écriture doit inscrire dans la mémoire le niveau de sensibilité du programme courant. Ainsi, si un programme souhaite écrire une constante de niveau public, il ne doit pas pouvoir le faire si son niveau de sensibilité est supérieur. Cela ressemble beaucoup à la labellisation, ce qui est normal puisque écrire un octet en mémoire revient à labelliser cet octet.

```
let write (r:ref{bounded (mkRef' r)}) (v:byte) (l:labels) : Memory (unit)
    (requires λ c0 → correctMemory c0.mem
        ∧ label_pre c0.ifc l)
    (ensures λ c0 x c1 → modifies c0.mem c1.mem [(mkRef' r)]
        ∧ correctMemory c0.mem
        ∧ Some? (get c1.mem (mkRef' r))
        ∧ (let Some c = (get c1.mem (mkRef' r)) in
            label_post c0.ifc c c1.ifc v l)
        ∧ c0.ifc == c1.ifc)
=
    let vl = label v l in
    write_raw r vl
```

Nous pouvons maintenant lire et écrire en mémoire. Toutefois, avant de décrire un exemple d'utilisation, nous présentons comment exécuter une fonction dans un environnement temporaire. Il s'agit d'une version de `toLabeled` pour notre gestionnaire de mémoire.

Exécution labellisée

Cette nouvelle fonction est l'exact équivalent de `toLabeled` pour le contexte CFI de notre gestionnaire de mémoire. Ainsi, nous pouvons exécuter une fonction sans qu'elle ait d'effet de bord sur le reste du contexte CFI du programme.

Cependant, nous avons besoin d'accéder au contexte CFI et ce via les primitives du contexte LIO. Pour répondre à ce besoin nous écrivons un accesseur et un mutateur qui retournent les champs du contexte de notre gestionnaire de mémoire. La partie délicate de ces fonctions dépend de la modélisation de l'état mémoire qui doit rester dans le monde de la vérification puisqu'il n'a pas de sens à l'exécution. Ainsi, notre accesseur `getMemorySTATE` retourne un couple de valeurs

dont le second élément ne sera jamais extrait en langage C. Il combine l'accesseur de LIO avec celui de notre effet `Memory`.

```
let getMemorySTATE (_:unit) : Memory (lioCTX * G.erased memory)
    (requires λ c0 → ⊤)
    (ensures λ c0 x c1 →  x == (c1.ifc, c1.mem)
        ∧ c0 == c1)
=
    let c0 =(MEMORY?.get ()) in
    let n0 = getSTATE () in
    assert(c0.ifc == n0);
    (n0, c0.mem)
```

Il est important de remarquer que nous vérifions que le contexte CFI de notre effet `Memory` est le même que celui de notre effet `Lio` – via `àssert(c0.ifc == n0)`. En effet, le contexte CFI de notre gestionnaire de mémoire (`c0`) ne sera pas extrait et donc ce test permet de montrer que les effets ont bien le même contexte.

De la même manière, nous pouvons écrire le mutateur `setMemorySTATE_unsafe` qui fera appel au mutateur non sûr de LIO. Pourquoi ne vérifions-nous pas la monotonie du contexte dans ce mutateur ? La réponse est que nous cassons cette monotonie dans le cadre de l'exécution labellisée.

```
let setMemorySTATE_unsafe (s:lioCTX * G.erased memory) : Memory (unit)
    (requires λ c0 → ⊤)
    (ensures λ c0 x c1 → s == (c1.ifc, c1.mem))
=
    setSTATE_unsafe (fst s);
    let c1 = ({ifc=(fst s); mem=(snd s)}) in
    MEMORY?.put (c1);
    ()
```

Dans ce mutateur, nous construisons un contexte en tenant compte de la mémoire. Une fois extrait, cet accesseur et ce mutateur ne feront que des appels vers les fonctions d'états de LIO – `getSTATE` et `setSTATE_unsafe`. Nous pouvons alors montrer le code de notre exécution labellisée qui est très similaire à celle de LIO. De fait, nous ne donnerons pas les pré- et post-conditions qui travaillent sur des `memorySTATE` au lieu de travailler sur des `lioCTX`.

```
let toLabeled_mem   #a
                #b
                (#pre)
                (#post)
                (l:labels)
                (c:clearance)
                ($call:((v:b) → Memory a (pre v) (post v)))
```

```
                (params:b)
        : Memory (labeled a)
    (requires λ c0 → toLabeled_mem_pre pre post c0 l c params)
    (ensures λ c0 x c1 → toLabeled_mem_post post c0 x c1 l c params)
=
    let (n0,m0) = getMemorySTATE () in          (* sauvegarde du contexte *)
    let r = (call params) in                    (* appel de la fonction *)
    let (n1,m1) = getMemorySTATE () in          (* sauvegarde du contexte *)
    let n2 = ({cur=n0.cur; cle=n1.cle}) in      (* création du contexte final*)
    let vl = label r l in                       (* labellisation du résultat *)
    setMemorySTATE_unsafe (n2,m1);              (* mise à jour du contexte *)
    vl                                          (* valeur retournée *)
```

Maintenant nous avons un gestionnaire de mémoire labellisée qui utilise LIO pour éviter les fuites d'informations. Il nous est possible de lire et d'écrire en mémoire. Nous pouvons simuler l'exécution non interruptible de plusieurs programmes grâce à l'exécution labellisée. Ainsi, nous illustrons dans la suite son utilisation par trois tâches qui accèdent à des adresses communes en mémoire.

8.3 Utilisation

Notre programme est un petit système embarqué avec trois tâches : un compteur, un capteur et un gestionnaire. Le compteur stocke et incrémente une valeur en mémoire. Le capteur écrit une valeur en fonction d'une entrée uniquement s'il est configuré par le gestionnaire pour le faire. Le gestionnaire a deux actions : il calcule une somme pour le contrôle du compteur et il calcule la moyenne des valeurs du capteur.

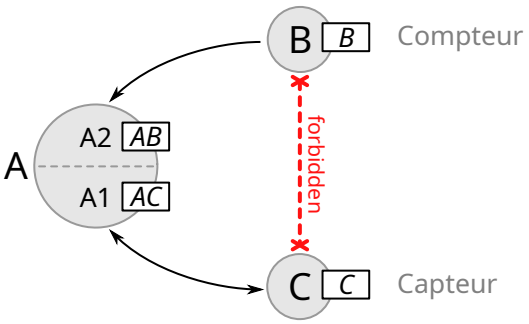

FIGURE 8.2 – Illustration du système et ses interactions

Ainsi, le gestionnaire communique avec toutes les tâches mais le compteur et le capteur ne doivent jamais échanger de l'information. Nous utilisons le nommage suivant : `B` pour le compteur, `C` pour le capteur et `A` pour le gestionnaire. Cependant, le gestionnaire sera divisé en

deux sous-tâches : `A1` qui traite avec le capteur et `A2` qui traite avec le compteur. La figure 8.2 montre les communications entre ces différentes tâches ainsi que l'échange de données interdit.

La tâche `C` reçoit également des données depuis la tâche `A` mais sans avoir d'informations de la tâche `A`. La tâche `B` ne fait que produire des informations à destination de la tâche `A` qui les lira par la suite. Ces tâches communiquent sur une mémoire partagée via le gestionnaire mémoire que nous avons décrit précédemment. Nous décrivons brièvement chaque tâche pour montrer comment utiliser à la fois notre gestionnaire de mémoire et LIO. Ainsi, la mémoire partagée possède une labellisation qui dépend des tâches qui y accèdent comme illustré sur la figure 8.3.

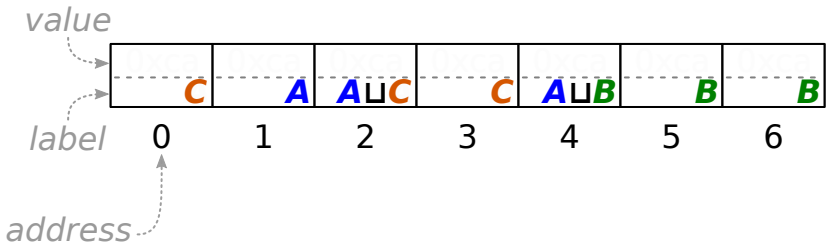

FIGURE 8.3 – Labellisation de la mémoire du système

Avant de décrire les différentes tâches, nous introduisons quelques prédicats utilitaires. En effet, ils servent à rendre les pré- et post-conditions plus lisibles. Le prédicat `hasValue` est satisfait si la valeur `v` est contenue dans la mémoire `m` à l'adresse `r`. Similairement, le prédicat `hasLabel` est satisfait pour le label `l` donné. Le label donné doit correspondre exactement avec celui de l'adresse donnée. Le prédicat `incr` vérifie qu'une valeur à cette adresse ne diffère numériquement que de 1 entre deux mémoires. Afin de nous concentrer sur LIO, nous ne tenons pas compte des dépassements de capacité dans ces exemples et donc 0 est bien supérieur à 255 dans ce cas.

```
unfold let hasValue m r v =
    correctMemory m  ∧ bounded r ∧ Some? (get m r)
    ∧ (let Some w = (get m r) in valueOf w == v)

unfold let hasLabel m r l =
    correctMemory m ∧ bounded r ∧ Some? (get m r)
    ∧ (let Some w = (get m r) in _labelOf w == l)

unfold let incr m0 m1 r =
    correctMemory m0 ∧ bounded r ∧ Some? (get m0 r)
    ∧ hasValue m1 r ((valueOf (Some?.v (get m0 r))) `U8.add_underspec` 1uy)

unfold let mcle c l : Type0 = ∀ (k:labels{contains k l}). c.cle k
```

Le dernier prédicat permet de simplifier la spécification d'une autorisation. En effet, nous pouvons écrire `mcle c0.ifc [A;C;AC]` au lieu d'écrire `c0.ifc.cle AC ∧ c0.ifc.cle C ∧ c0.ifc.cle A`. Nous faisons remarquer que les précédents prédicats possèdent de l'attribut `unfold`. Cela sert à "copier/coller" la définition un peu comme une macro en langage C. Grâce à cela, F⋆ peut facilement réduire les préconditions les plus faibles et donc alléger la vérification.

8.3.1 Tâche A

Cette tâche – appelée gestionnaire – communique avec les tâches `B` et `C`. Nous divisons cette tâche en deux sous-tâches :

— `A1` qui active le capteur (tâche `C`) en mettant la valeur de l'adresse 0 à une valeur constante, puis calcule la valeur moyenne du capteur. Le calcul de la moyenne s'effectue par une accumulation des valeurs du capteur dans l'adresse 2 et le nombre de valeurs accumulées est stocké dans l'adresse 1.

— `A2` qui calcule une somme de contrôle du compteur (tâche `B`). Cette somme est très simple puisqu'il s'agit d'une simple addition de la partie haute et de la partie basse du nombre. En effet, le compteur est encodé sur 16 bits soit deux octets qui représentent les parties haute et basse.

La première des sous-tâches, que nous appelons `programA1`, vérifie le label des adresses mémoire qu'elle utilise via `hasLabel`. La précondition assure que notre label courant est à `Nothing`, ce qui est essentiel pour configurer le capteur sur l'adresse 0 – après la lecture de l'adresse 1, il ne sera plus possible d'écrire au niveau de `C` mais uniquement au niveau de `AC`. Or comme présenté sur la figure 8.3, nous ne voulons pas que l'adresse 0 puisse être contaminée avec une connaissance venant des adresses de la tâche `A`. La postcondition assure que le label courant est à `AC`, ce qui indique que la tâche a lu des informations de `A` et `C`. De plus, la postcondition assure que la valeur de l'adresse 1 est incrémentée de 1 et que la valeur de l'adresse 2 est additionnée avec la valeur de l'adresse 3. Pour des raisons de simplicité nous acceptons les dépassements de capacité dans les nombres d'où l'utilisation d'un opérateur d'addition spécial (`add_underspec`). Nous décrivons ci-dessous le code de `programA1`.

```
let programA1  (_:unit) : Memory (unit)
    (requires λ c0 →  correctMemory c0.mem
        ∧ c0.ifc.cur = Nothing
        ∧ mcle [A;C;AC]
        ∧ hasLabel c0.mem 1 A ∧ hasLabel c0.mem 2 AC ∧ hasLabel c0.mem 3 C)
    (ensures λ c0 x c1 → correctMemory c1.mem
        ∧ modifies c0.mem c1.mem [1;2;0]
        ∧ incr c0.mem c1.mem 1
        ∧ Some? (get c0.mem 3) ∧ Some? (get c0.mem 2)
        ∧ Some? (get c1.mem 2) ∧ Some? (get c1.mem 0)
        ∧ (valueOf (Some?.v (get c1.mem 2)))
```

```
                = (valueOf (Some?.v (get c0.mem 2)))
                    `U8.add_underspec` (valueOf (Some?.v (get c0.mem 3)))
        ∧ hasLabel c1.mem 0 C ∧ hasLabel c1.mem 1 A ∧ hasLabel c1.mem 2 AC
        ∧ c0.ifc.cle == c1.ifc.cle
        ∧ c1.ifc.cur = AC)
=
    write 0ul (0uy) C;
    write 1ul ((read 1ul) `U8.add_underspec` 1uy) A;
    write 2ul ((read 2ul) `U8.add_underspec` (read 3ul)) AC
```

Nous faisons remarquer que nous devons vérifier tous les labels autorisés possibles – ici `A`, `C` et `AC`. Cela est dû à la généricité de notre autorisation. En effet, il est possible d'exprimer une autorisation qui refuse juste le label `A` et accepte le label `AC`. Cela n'a pas de sens au niveau du CFI mais rend la définition de l'autorisation beaucoup plus simple. Dans un développement futur, il serait parfaitement possible de contraindre la définition de l'autorisation afin de réduire le coût des pré- et post-conditions.

La seconde des sous-tâches, que nous appelons `programA2`, calcule la somme de contrôle – une simple addition – des adresses de la tâche `B`. Tout comme pour `programA1`, la précondition assure que notre programme est autorisé à accéder à des informations labellisées `B` et `AB`. Nous soulignons que le label courant n'est pas nécessairement fixé mais doit être inférieur ou égal à `A` (`c0.ifc.cur` ⊑ `A`).

```
let programA2 (_:unit) : Memory (unit)
    (requires λ c0 → correctMemory c0.mem
        ∧ c0.ifc.cur ⊑ A
        ∧ mcle [AB;B]
        ∧ hasLabel c0.mem 5 B ∧ hasLabel c0.mem 6 B)
    (ensures λ c0 x c1 → correctMemory c1.mem
        ∧ modifies c0.mem c1.mem [4]
        ∧ Some? (get c0.mem 5) ∧ Some? (get c0.mem 6) ∧ Some? (get c1.mem 4)
        ∧ (valueOf (Some?.v (get c1.mem 4)))
            = (valueOf (Some?.v (get c0.mem 5)))
                `U8.add_underspec` (valueOf (Some?.v (get c0.mem 6))))
=
    write 4ul ((read 5ul) `U8.add_underspec` (read 6ul)) AB;
    ()
```

Le code de cette fonction est finalement assez simple et nous voyons dans ces exemples une caractéristique d'un tel langage formel : la verbosité de la spécification. Cette caractéristique rend des langages comme F⋆ difficiles à utiliser, mais cela les rend aussi extrêmement précis. Dans `programA2`, nous connaissons exactement le comportement de la fonction, par exemple l'impact en mémoire uniquement sur l'adresse 4 (`modifies c0.mem c1.mem [4]`).

8.3.2 Tâche B

Cette fonction incrémente de 1 un compteur encodé sur 16 bits. L'algorithme est très simple : si la partie basse est égale à 255, elle est remise à 0 et la partie haute est incrémentée, sinon la partie basse est incrémentée de 1. Nous rappelons que cet algorithme n'est pas sûr en termes de dépassement : nous utilisons l'addition non vérifiée.

Notre tâche lit et modifie potentiellement les adresses 5 et 6 qui doivent être de niveau `B`. Elle est uniquement autorisée à manipuler des données de niveau `B`. C'est pourquoi nous devons assurer que la tâche `A` ne contamine pas cette tâche.

```
let cle_B (l:labels) = l ⊑ B

let programB (_:unit) : Memory (unit)
    (requires λ c0 → correctMemory c0.mem
        ∧ c0.ifc.cle == cle_B ∧ c0.ifc.cur = Nothing
        ∧ hasLabel c0.mem 5 B ∧ hasLabel c0.mem 6 B)
    (ensures λ c0 x c1 → correctMemory c1.mem
        ∧ modifies c0.mem c1.mem [5;6]
        ∧ hasLabel c1.mem 5 B ∧ hasLabel c1.mem 6 B
        ∧ (
            (hasValue c0.mem 5 255uy
                ⟹ (hasValue c1.mem 5 0uy ∧ incr c0.mem c1.mem 6))
            ∨ ( (getValue c0.mem 5) ≠ 255uy
                ⟹ ( incr c0.mem c1.mem 5 )))
        ∧ c1.ifc.cle == c0.ifc.cle ∧ c1.ifc.cur = B)
=
    if (read 5ul) = 255uy then (
        write 6ul ((read 6ul) `U8.add_underspec` 1uy) B;
        write 5ul 0uy B
    )else (
        write 5ul ((read 5ul) `U8.add_underspec` 1uy) B
    )
```

Nous présentons des préconditions précises sur l'état des labels dans la mémoire. Il est évident qu'avant d'appeler chacune des tâches notre système doit être initialisé. Nous donnons ci-dessous une fonction d'initialisation de la tâche `B`.

```
let initmemoryB (_:unit) : Memory (unit)
(requires λ c0 → correctMemory c0.mem
    ∧ c0.ifc.cle B ∧ c0.ifc.cur ⊑ B)
(ensures λ c0 x c1 → correctMemory c1.mem
    ∧ hasLabel c1.mem 5 B   ∧ hasLabel c1.mem 6 B
    ∧ hasValue c1.mem 5 0uy ∧ hasValue c1.mem 6 0uy
    ∧ modifies c0.mem c1.mem [5;6]
```

```
    ∧ c0.ifc == c1.ifc)
=
    write 5ul 0uy B;
    write 6ul 0uy B
```

Dans la fonction `initmemoryB`, l'état de la mémoire aux adresses 5 et 6 est considéré comme inconnu à l'appel de la fonction. Cependant, l'initialisation des parties haute et basse de notre compteur au label `B` permet d'assurer deux points : le label de la mémoire et une valeur initiale. Cette fonction d'initialisation a un impact uniquement sur la mémoire.

8.3.3 Tâche C

La lecture d'un capteur se fait via une fonction d'entrée/sortie, mais pour la vérification de cet exemple nous utilisons `magic` qui permet de choisir un terme quelconque qui correspond au type attendu. Cependant, pour l'extraction en C, nous remplaçons l'appel à `magic` par une constante arbitrairement choisie à 0.

```
let programC (_:unit) : Memory (unit)
    (requires λ c0 → correctMemory c0.mem
        ∧ c0.ifc.cle C ∧ c0.ifc.cur ⊑ C
        ∧ hasLabel c0.mem 3 C ∧ hasLabel c0.mem 0 C)
    (ensures λ c0 x c1 → correctMemory c1.mem
        ∧ modifies c0.mem c1.mem [3]
        ∧ hasLabel c1.mem 3 C
        ∧ c1.ifc.cur = C ∧ c1.ifc.cle == c0.ifc.cle)
=
    if (read 0ul) = 0uy then
        write 3ul (magic ()) C
    else
        ()
```

Tout comme les tâches précédentes, cette tâche n'utilise que deux adresses (0 pour la configuration et 3 pour l'écriture de la valeur du capteur). À la fin de l'exécution de la tâche, le label courant est au niveau `C`, ce qui indique que la tâche n'a pas pu écrire plus haut que ce que l'autorisation `c0.ifc.cle` ne permet.

8.3.4 Extraction

Nous avons vu comment implémenter et utiliser un gestionnaire mémoire et un programme d'exemple. Cependant, à quoi ressemble le code extrait en C ? Nous prendrons ici la tâche `B` définie précédemment et nous montrons son extraction en langage C :

```
void programB() {
    if (read((uint32_t)5U) == (uint8_t)255U) {
```

```
        write((uint32_t)6U, read((uint32_t)6U) + (uint8_t)1U, B);
        write((uint32_t)5U, (uint8_t)0U, B);
    } else
        write((uint32_t)5U, read((uint32_t)5U) + (uint8_t)1U, B);
}
```

Nous voulons faire remarquer que les deux codes sont très proches. À l'exception de la spécification et quelques notations mineures, les codes sont similaires. Cette ressemblance garantit la lisibilité du code extrait. Ainsi, comment sont extraites les primitives ?

Dans la primitive `read` du gestionnaire mémoire, nous lisons la valeur labellisée stockée à l'adresse `r` et ensuite nous retournons cette valeur. Le fait que `read_raw` ne manipule pas les valeurs labellisées rend la vérification plus simple puisque les valeurs labellisées seront considérées comme inchangées par le code C.

```
uint8_t read(uint32_t r) {
    cell lv = read_raw(r);
    return unlabel__uint8_t(lv);
}
```

La primitive `unlabel` manipule l'état du CFI. Nous récupérons le contexte courant pour le mettre à jour. Dans le code ci-dessous nous pouvons voir le label courant `cur` être augmenté via l'opérateur `join` (équivalent à l'opérateur $\sqcup$).

```
uint8_t unlabel__uint8_t(cell vl) {
    lioSTATE c0 = getSTATE((void *)(uint8_t)0U);
    setSTATE(((lioSTATE){
        .cur = join(c0.cur, _labelOf__uint8_t(vl)),
        .cle = c0.cle }));
    return vl.value;
}
```

Notre exemple présente un cas particulier : les labels ne sont pas dynamiques. En effet, les labels sont fixés à la conception comme présenté dans la figure 8.3. Nous devons nous interroger sur l'impact des labels sur un tel système : puisque F$\star$ peut valider notre programme, que se passe-t-il si nous supprimons les labels à l'exécution ? Avons-nous intérêt à le faire ?

8.4 Labellisation statique

Précédemment, nous avons présenté le code en C de la fonction `unlabel`. Ce code effectue deux opérations sur le contexte avant de retourner la valeur de la cellule. Lorsque le programme est sûr pour chaque entrée du point de vue du CFI, ces opérations ont un coût qui n'est pas

nécessaire. En effet, si un programme manipule uniquement des informations de niveau secret, il ne sert à rien de mettre à jour le contexte du CFI à secret à chaque opération.

8.4.1 Changements

En F⋆, nous pouvons utiliser le type `G.erased` qui indique que le code ne sera pas extrait (voir chapitre 2). Notre objectif étant de vérifier le programme en termes de CFI, mais d'enlever la labellisation à l'exécution, nous devons supprimer le contexte. L'utilisation du type `G.erased` nous permet d'y parvenir comme montré dans le code ci-dessous. Dans ce code, nous indiquons que `lioCTX` ne peut être extrait en changeant sa définition :

```
type lioCTX = G.erased (lioSTATE)
```

Il serait possible de nous demander pour quelle raison nous n'avons pas ajouté l'attribut `noextract`. La raison est sémantique puisque dans le cas de `noextract`, F⋆ n'extraira pas `lioCTX` mais il validera tout code qui le manipule, ce que nous ne souhaitons pas – les appels `getSTATE` et `setSTATE_unsafe` sont conservés. Par ailleurs, nous remplaçons les accesseurs et les mutateurs de l'effet par leur version effaçable. Par exemple, au lieu de faire appel à `getSTATE`, nous faisons appel à `LIO?.get`. Que ces fonctions soient effacées à la compilation est important puisqu'une version statique de LIO ne modifie pas le contexte du CFI à l'exécution.

Les valeurs labellisées n'ont plus vraiment de sens une fois la vérification terminée. Le contexte du CFI étant effacé, les labels n'ont plus d'utilité. Nous les supprimons de la même manière que le contexte – c'est-à-dire en les rendant fantomatiques.

```
noeq
type labeled (a:Type) = {value:a; tag:G.erased labels}

let _labelOf (#a:Type) (vl:labeled a) : Ghost (labels)
    (requires ⊤)
    (ensures λ x → x == G.reveal vl.tag)
=
    G.reveal vl.tag

let labelOf (#a:Type) (vl:labeled a) : Ghost (labels)
    (requires ⊤)
    (ensures λ x → x == rev vl.tag)
=
    _labelOf vl
```

La primitive `labelOf` change d'effet : elle devient fantomatique puisqu'elle parle des labels des valeurs labellisées. Nous utilisons `G.reveal` pour accéder à une information fantomatique. Dans la grande majorité de notre bibliothèque CFI, le code reste inchangé – à l'exception des mutateurs et des accesseurs – comme illustré par la fonction `unlabel` :

```
unfold let unlabel_pre c0 vl = c0.cle (c0.cur ⊔ _labelOf vl)
unfold let unlabel_post c0 x c1 vl =  x == vl.value
    ∧ c1 == {cur=(c0.cur ⊔ _labelOf vl); cle=c0.cle}
    ∧ monotonicity c0 c1

let unlabel (#a:Type) (vl:labeled a)  : Lio (a)
    (requires λ c0 → unlabel_pre c0 vl)
    (ensures λ c0 x c1 → unlabel_post c0 x c1 vl)
=
    let c0 = LIO?.get () in
    LIO?.put ({cur=(c0.cur ⊔ _labelOf vl); cle=c0.cle});
    vl.value
```

À l'exception de `labelOf`, les signatures des primitives de la bibliothèque LIO ne changent pas, nous pouvons utiliser tout programme dont l'exécution ne dépend pas des labels – ce qui correspond à notre exemple. Cependant, un programme qui labellise selon une entrée de l'utilisateur ne peut pas être rendu statique sauf si nous connaissons cette entrée à l'avance ou si nous la contrôlons dynamiquement. Dans le code ci-dessous, nous illustrons la transformation d'un programme qui labellise un nom avec le label de l'utilisateur courant (les deux labels possibles sont `Admin` et `Nobody`) :

```
(* UserLabels = | Admin | Nobody *)

let dynamic =
    let vl = label (getName ()) (getUserLabel ()) in
    ...

let statified =
    let vl = match (getUserLabel ()) with
        | Admin → label n (G.hide Admin)
        | _ → label n (G.hide Nobody)) in
    ...
```

La fonction `dynamic` labellise directement avec les entrées. Cependant, une version statique doit transformer un label en une information statique à la compilation. Cela est réalisé à l'aide des branchements comme illustré par la fonction `statified` mais nécessite dans le pire scénario de tester l'intégralité des labels.

Notre exemple n'ayant pas de tests dépendant des labels à l'exécution, nous pouvons comparer directement la différence entre un code statique et dynamique pour connaître le coût réel des labels dynamiques. Bien qu'il aurait été intéressant d'approfondir le coût engendré par une conversion automatique depuis un code dynamique dépendant des labels à l'exécution vers un code statique, nous n'abordons pas cette question dans notre travail. Nous nous contentons de comparer un code dynamique avec un code statique.

8.4.2 Extraction

À quoi ressemble le code généré avec une labellisation statique ? Dans ce paragraphe, nous présentons les changements entre la version statique et la version dynamique. En effet, les informations de la vérification sont effacées à la compilation – certaines n'ont pas de sens en termes d'exécution. Pour illustrer ce changement, nous allons reprendre la fonction `read` que nous présentons pour l'extraction de notre LIO minimal.

```
/* version dynamique */
uint8_t read(uint32_t r) {
    cell lv = read_raw(r);
    return unlabel__uint8_t(lv);
}

/* version statique */
uint8_t read(uint32_t r) {
    uint8_t lv = read_raw(r);
    return lv;
}
```

Nous portons l'attention sur la disparition de la notion de CFI. Cela vient du fait que le contexte et les appels aux fonctions de mise à jour et lecture du contexte sont effacés. De fait, le code est considéré comme n'ayant aucun impact et donc il n'est pas extrait. Nous pouvons montrer l'impact de ce changement sur le type `cell` qui est un couple de la forme ⟨`value,label`⟩ et qui s'extrait de manière très différente :

```
/* version dynamique */
typedef struct cell_s {
    uint8_t value;
    labels tag;
} cell;

/* version statique */
typedef uint8_t cell;
```

C'est pourquoi la fonction `read` peut stocker une valeur de `read_raw` qui est de type `cell` dans un type `uint8_t`. Cependant, nous ne répondons pas complètement à la question à l'origine d'une version statique : avons-nous intérêt à supprimer les labels ? Si nous avons montré que le code est plus léger, nous présentons dans le paragraphe suivant la différence en termes d'exécution.

8.5 Mesures

Nous avons vu que le code est plus petit en version statique. Nous allons présenter le coût en mémoire et la vitesse d'exécution. Pour le coût mémoire, nous avons déjà les éléments pour le calculer. Dans les cellules mémoire, les labels ont un coût de 1 octet comme la valeur qu'elles contiennent. Ainsi, une cellule labellisée coûtera 2 octets dans la version dynamique, en supprimant le label elle ne coûtera qu'un seul octet.

Cela permet de calculer la taille du tableau nécessaire pour le gestionnaire de mémoire (`hwmem`) qui contient `0x400` cellules – soit 1024 cellules en décimal. Dans la version dynamique, il faudra compter 2048 octets contre 1024 pour la version statique. Nous vérifions ces chiffres avec l'opérateur **`sizeof`** du langage C.

```
/* version dynamique */
sizeof(cell)    == 2
sizeof(hwmem)   == 2048

/* version statique */
sizeof(cell)    == 1
sizeof(hwmem)   == 1024
```

Nous faisons remarquer que le gain est de 50 % dans notre exemple. Quelle que soit l'implémentation, la version dynamique sera toujours plus lente que la version statique. Cela se remarque aussi sur une implémentation matérielle (Ferraiuolo et al. 2018). Nous soulignons que si les labels coûtent plus d'un octet, cela augmentera l'efficacité de l'approche statique. Inversement, si nous augmentons la capacité des valeurs labellisées – ici les cellules mémoire – l'efficacité du statique sera moindre mais toujours supérieure. Nous illustrons les résultats pour notre programme dans le graphique 3 de la figure 8.4.

Pour mesurer le temps d'exécution, nous mesurons le temps de la tâche `B` qui est un compteur. Notre programme de test exécute cette fonction 100 millions de fois afin de mesurer la vitesse d'exécution d'une version. Nous avons alors deux programmes, l'un dynamique et l'autre statique. Après avoir exécuté ces deux programmes sur notre machine de test, nous obtenons 11,1 secondes pour le programme dynamique et 4 secondes pour le programme statique. Nous résumons ces résultats dans le graphique 4 de la figure 8.4.

Nous mesurons également la taille du code. Comme vu précédemment, le code extrait n'est pas le même entre la version dynamique et la version statique. Cela conduit à une réduction du code – voir le graphique (1) de la figure 8.4 – et explique ainsi les gains obtenus à l'exécution. Nous n'incluons pas le nombre de lignes de l'extraction du treillis – 766 lignes de code en C – puisque ce dernier est identique pour les deux versions. En effet, un programme dépendant des labels a besoin du code qui définit le treillis et les labels indépendamment du moment – statique ou dynamique – de la labellisation.

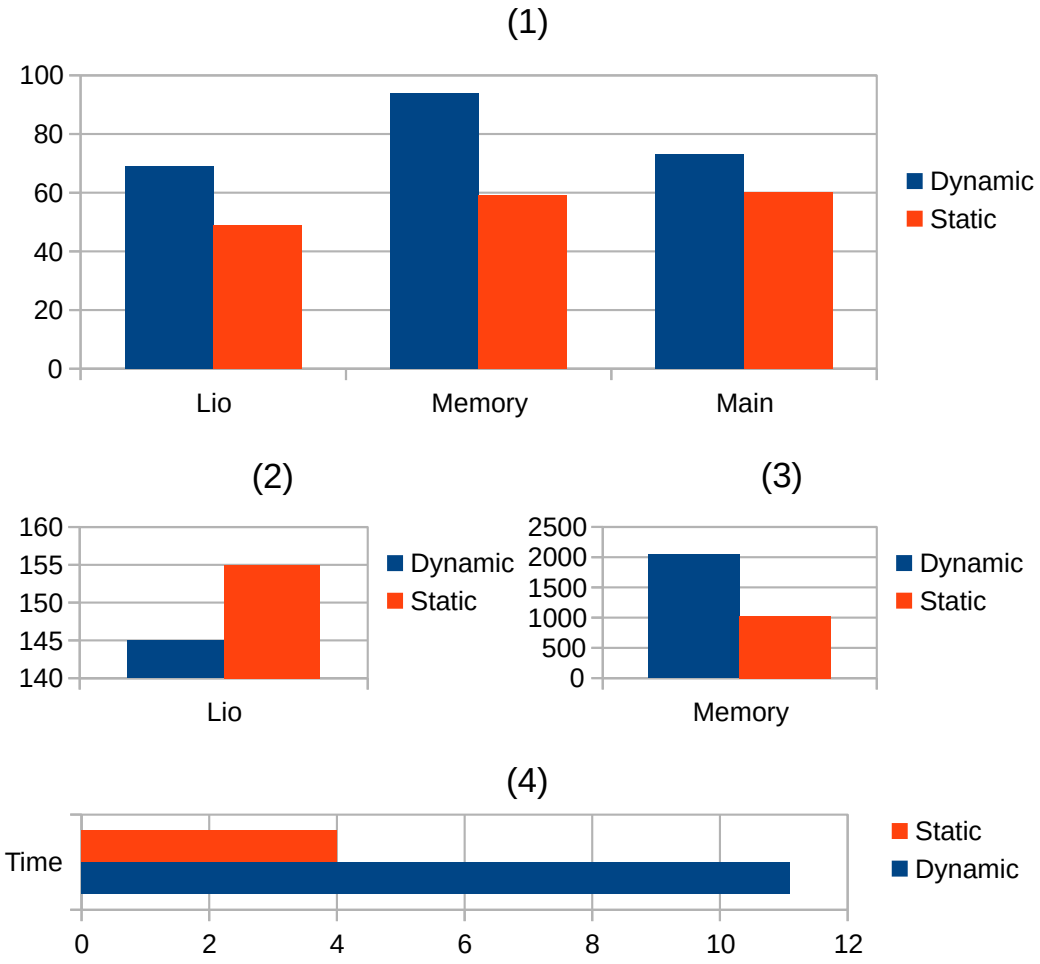

FIGURE 8.4 – Titre des graphiques : (1) nombre de lignes de code en C ; (2) nombre de lignes de code en F⋆ ; (3) utilisation mémoire ; (4) temps d'exécution

Contrairement aux autres résultats de la figure 8.4, le code statique en F⋆ est plus lourd – graphique (2) de la figure 8.4 –, ce qui s'expliquerait : 1/ par la nécessité de préciser la preuve pour montrer l'inutilité des tests à l'exécution et 2/ par la spécification nécessaire pour rendre le contexte inexistant à l'exécution. À cause de la simplicité de notre exemple, nous avons seulement la seconde option. Toutefois, nous retrouvons dans notre travail sur LIO⋆ ces deux raisons qui alourdissent le code statique en F⋆ : plus les tests à l'exécution sont supprimés, plus l'effort de preuve sera important. Inversement, plus les tests sont réalisés à l'exécution, moins l'effort de preuve sera important. Nous faisons remarquer que le calcul de la précondition la plus faible nous a permis de réduire cette différence grâce à la richesse des connaissances sur le programme.

8.6 Discussions

Dans ce chapitre, nous avons présenté un gestionnaire de mémoire qui utilise LIO pour protéger deux entrées/sorties : la lecture et l'écriture. Ainsi, nous avons montré comment implémenter

le CFI et comment l'utiliser. Nous sommes allés plus loin en proposant une solution dynamique et une solution statique.

Pouvons-nous affirmer que LIO protège complètement contre la fuite d'informations ? La réponse n'est pas simple : si on tient compte uniquement des entrées/sorties alors notre programme est protégé contre la fuite d'informations des flux explicites ou implicites ; si on tient compte des canaux cachés alors notre programme n'est pas protégé. Cela peut être affirmé au moyen d'un contre-exemple assez simple : au lieu de stocker l'information dans des valeurs labellisées, un attaquant peut les stocker dans la valeur de l'adresse. Nous illustrons ce cas avec le code ci-dessous :

```
(* le label courant est PUBLIC *)
write 2ul 0uy;
write 3ul 0uy;

let pwd = read 5ul in
(* le label courant est SECRET *)

if (pwd == "secret") then
    write 3ul 0uy
else
    write 2ul 0uy
```

Si dans un programme l'attaquant initialise deux adresses à un label de sensibilité publique, alors il peut écrire un booléen en mémoire avec les adresses et utiliser une autre tâche moins privilégiée pour récupérer l'information. Nous ne ciblons pas ce type d'attaque mais le MMU que nous avons présenté au chapitre 6 à la section 6.2 résout partiellement ce problème.

En effet, la gestion de pages virtuelles rend les adresses propres à chaque tâche ou groupe de tâches. De fait, le code présenté ci-dessus est plus limité : il ne peut faire évader l'information qu'avec des tâches autorisées à partager la mémoire. Ceci dit, il existe d'autres formes de canaux cachés comme souligné par l'article (Sabelfeld et Myers 2003). Il serait très intéressant de combiner le gestionnaire mémoire et le MMU décrit précédemment, nous laissons ce travail futur comme une ouverture que nous n'aborderons pas par manque de temps.

Nous avons présenté deux approches du CFI, l'approche dynamique et l'approche statique. Bien que nous ayons montré les avantages de chacune d'entre elles, serait-il intéressant de combiner les deux ? En effet, l'effort de preuve peut être très lourd en cas de manipulation de données interprétées à l'exécution – par exemple des scripts fournis par l'utilisateur – et dans d'autres cas très simples – les données peuvent être constantes par exemple. C'est ce que nous allons montrer avec un projet – appelé LIO⋆ – qui permet de mélanger différentes approches de CFI.

Approche hybride

Nous avons présenté dans le précédent les avantages et les inconvénients d'un CFI statique et d'un CFI dynamique. Nous nous sommes alors posé la question des bénéfices qu'une combinaison des deux pourrait apporter.

> Nous présentons une version préliminaire de LIO⋆ dans le papier (Marty et al. 2020) où nous avons trois déclinaisons de LIO – une version dynamique qui peut échouer en cas d'exception, une version dynamique qui n'échoue pas (correspond à l'approche dynamique présentée au chapitre 7) et une version statique qui n'échoue pas (correspond à l'approche statique présentée en §8.4). Dans le papier (Marty et al. 2020), nous ne montrons pas comment combiner ces versions, qui sont utilisées comme bibliothèques indépendantes. Dans l'état actuel de LIO⋆ (qui reste un travail en cours), nous avons une bibliothèque qui permet de combiner les trois versions dans un même programme F⋆.

Dans ce chapitre, nous présentons une version jouet de LIO⋆. Cette version a pour objectif de mettre en avant les différences avec LIO et de montrer comment combiner les approches statiques et dynamiques afin de pouvoir passer de l'une à l'autre au sein d'un même programme. Nous décrivons ainsi une implémentation, une manière de prouver cette implémentation et les différences entre le travail en cours et ce document.

9.1 Implémentation

Pourquoi avons-nous intérêt à avoir des parties d'un même programme avec un CFI statique et d'autres avec un CFI dynamique ? Dans le papier (Buiras, Vytiniotis et Russo 2015), les auteurs proposent de supprimer des tests redondants afin d'alléger le coût à l'exécution du programme. Par exemple dans le code ci-dessous, l'addition de valeurs labellisées `idontknow` doit être impérativement dynamique puisque nous ne connaissons pas à l'avance toutes les valeurs labellisées. Toutefois, l'addition de valeurs labellisées `iampublic` de contexte *public* instancie l'addition et rend la labellisation redondante.

```
let add (vl1 vl2:labeled int) : Lio (int)
    (requires ...) (ensures ...)
=
    let v1 = unlabel vl1 in
```

```
    let v2 = unlabel vl2 in
    v1 + v2

let iampublic   =   (* fonction dans un contexte public *)
    let x = ⟨3,Public⟩ in
    let y = ⟨5,Public⟩ in
    print (add x y)

let idontknow y = (* fonction dans un contexte public *)
    let x = ⟨3,Public⟩ in
    print (add x y)
```

Dans notre exemple ci-dessus, une vérification statique du CFI de `iampublic` rendra le programme plus léger. Cependant dans le programme `idontknow`, la labellisation est nécessaire puisqu'elle peut modifier l'état du CFI. Contrairement à la fonction `iampublic`, le programme `idontknow` sera plus simple à vérifier dynamiquement si `y` ne peut être connu à la compilation.

Nous souhaitons avoir un programme qui peut à tout moment passer d'une vérification dynamique vers une vérification statique. Nous implémentons alors une version statique de LIO que nous appelons `SLio` puis une version dynamique que nous appelons `DLio`.

9.1.1 Superposition entre statique et dynamique

Dans le papier (Buiras, Vytiniotis et Russo 2015), les auteurs montrent comment utiliser le typage et une fonctionnalité du compilateur Glasgow Haskell Compiler (GHC) qui décharge les contraintes statiques par des tests dynamiques, et inversement, peut décharger des tests dynamiques par des contraintes statiques. Cela leur permet de passer d'une approche statique à une approche dynamique. Dans notre travail, nous avons implémenté une connexion entre un effet CFI dynamique et un effet CFI statique en F$\star$ qui, contrairement à l'article (Buiras, Vytiniotis et Russo 2015), ne nous permet pas de vérifier une fonction statiquement ou dynamiquement. Dans notre cas, le code est soit vérifié dynamiquement, soit vérifié statiquement, et nous montrons comment interconnecter les deux formes de vérification entre elles.

Imaginons une fonction vérifiée dynamiquement `f` qui appelle une fonction `g` qui sera vérifiée statiquement. La fonction `f` pourra exceptionnellement utiliser la primitive `getSTATE`, et le code de `g` n'importe pas dans cet exemple puisque seule l'évolution du label courant de `g` nous intéresse. En effet, l'appel à `g` peut modifier le label courant d'une manière statique, comme illustré sur la figure 9.1. Cela ne peut pas être pris en compte par le CFI dynamique. De fait, nous devons "lier" les deux mondes par une approximation du label courant. Pour cela, nous devons connecter les deux effets entre eux. De la même manière que nous avons connecté l'effet `LIO` à l'effet `PURE` – ici `SLIO` est également connecté à l'effet `PURE` –, nous connectons l'effet `DLIO` à l'effet `SLIO`. Cela permettra d'utiliser LIO statique dans LIO dynamique.

Notons que lors du retour à une vérification dynamique, le label courant doit être mis à jour

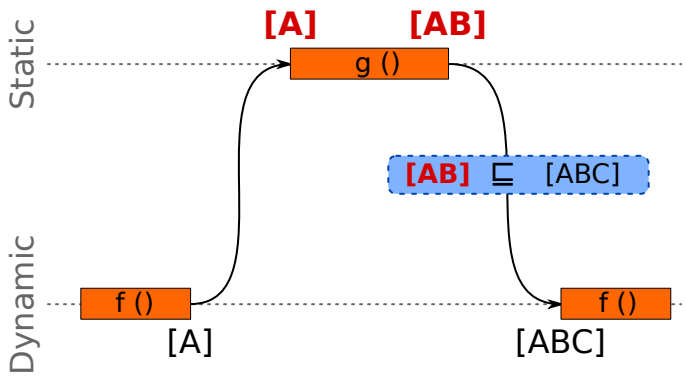

FIGURE 9.1 – Appel d'une fonction statiquement vérifiée dans une fonction vérifiée dynamiquement

dynamiquement pour refléter son évolution dans la partie statique. En effet, si la fonction `g` change le label courant, cette modification ne sera pas reflétée dynamiquement. Considérons le code suivant :

```
val g : (_:unit) → SLio (unit)
    (requires λ c → c.cur = Nothing)
    (ensures λ c0 x c1 → c1.cur = AB ∧ c1.cle == c0.cle)

let f (_:unit) : DLio (I32.t)
    (requires λ c0 → c0.cur = Nothing ∧ c0.cle AB)
    (ensures λ c0 x c1 → ⊤)
=
    g ();
    let c0 = getSTATE () in
    if c0.cur = AB then
        1l
    else
        0l
```

Nous faisons remarquer que la fonction `g` a un impact sur le contexte au niveau de la vérification mais pas au niveau du code généré. Il s'agit d'une désynchronisation du contexte vérifié avec celui de l'exécution. Ainsi, si nous exécutons le code C extrait de `f`, nous voyons que `c0.cur` ne sera pas égal à `AB` mais à `Nothing` et donc `f` retournera `0`.

```
int32_t f() {
    g();
    lioSTATE c0 = getSTATE((void *)(uint8_t)0U);
    if (__eq__LioLattice_labels(c0.cur, AB))
        return (int32_t)1;
```

```
    else
        return (int32_t)0;
}
```

Pour éviter cette situation, nous devons empêcher le retour à la vérification dynamique sans mettre à jour le label courant. Il s'agit de l'étape d'approximation entre le statique et le dynamique sur la figure 9.1. Nous rajoutons une primitive `raise` qui élève dynamiquement le label courant à la hauteur souhaitée tout en respectant la monotonie du contexte courant. Cela nous permet de corriger `f` comme montré ci-dessous :

```
let f (_:unit) : DLio (I32.t)
    (requires λ c0 → c0.cur = Nothing ∧ c0.cle AB)
    (ensures λ c0 x c1 → ⊤)
=
    g ();
    raise AB;
    let c0 = getSTATE () in
    if c0.cur = AB then
        1l
    else
        0l
```

Dans cette nouvelle version le code généré se comportera comme attendu et retournera la valeur correcte (c'est-à-dire 1). Cependant, nous devons assurer que la fonction `raise` est appelée systématiquement avant d'utiliser dynamiquement le contexte CFI. Il existe plusieurs manières de l'implémenter. L'une d'entre elles est d'utiliser des fonctions d'élévation que l'utilisateur fournit à chaque passage entre la vérification statique et la vérification dynamique.

Dans notre cas, nous utiliserons pour cet exemple une fonction d'élévation automatique. Cela explique pourquoi notre modélisation notera le contexte comme désynchronisé lors d'un changement entre la vérification statique et la vérification dynamique. Nous utilisons cette méthode pour des raisons techniques – à l'heure où nous écrivons ce document, l'extraction n'est pas complètement garantie – et pédagogiques – nous devrions redéfinir autrement les effets. En effet, dans nos travaux nous avons utilisé une autre approche que nous décrivons à la section 10.4. Dans ce chapitre, nous montrons une implémentation possible, puisque l'objectif est de montrer les avantages et inconvénients d'une interaction entre approche statique et dynamique.

9.1.2 Adaptation du code

Cette version est très similaire aux versions présentées précédemment dans le chapitre 7. Nous présentons ici les principales différences. Par exemple, le contexte `lioSTATE` est enrichi d'une information (`sync`) qui détermine s'il est synchronisé ou non avec l'état du contexte réel

– c'est-à-dire celui en mémoire à l'exécution. Nous donnons ci-dessous le nouveau contexte `lioSTATE`.

```
noeq
type lioSTATE = {
    cur:labels;
    cle:clearance;
    sync:G.erased bool
}
```

Nous définissons deux effets `SLIO` et `DLIO` comme défini dans la section précédente. Après avoir fait une élévation de `SLIO` vers `PURE`, nous élevons `DLIO` dans `SLIO`. Toutefois, contrairement aux autres élévations que nous avons montrées, nous modifions explicitement le contexte en mettant à jour l'état de synchronisation du contexte : celui-ci sera désormais non-synchronisé.

```
unfold let slio_wp (a: Type) = G.erased lioSTATE → SLIO?.post a → Type0
unfold let dlio_wp (a: Type) = lioSTATE → DLIO?.post a → Type0

val lift_slio_dlio : (a:Type) → (wp:slio_wp a) → dlio_wp a
let lift_slio_dlio (a:Type) (wp:slio_wp a) x p
    = wp x (λ (a,c) → p (a, G.reveal ({c with sync=false})))
sub_effect SLIO ⇝ DLIO = lift_slio_dlio
```

La fonction d'élévation ci-dessus est peu plus complexe que celles que nous avons montrées jusqu'ici. En effet, nous montrons ici comment transformer les préconditions les plus faibles de `SLIO` vers celles de `DLIO`. Notons que le contexte CFI dans `SLIO` est de type effacé (`G.erased lioSTATE`). Pour transformer ce dernier en contexte `DLIO`, nous devons le rendre visible à nouveau – c'est-à-dire le faire revenir depuis le monde effacé. Comme cette transformation est interdite, nous donnons un contexte qui est une approximation et nous montrons qu'elle est équivalente. En effet, une information fantomatique – de type `G.erased` – ne peut pas être utilisée dans le programme : nous sommes au niveau de la précondition la plus faible, ce qui ne peut pas être manipulé par le programme.

Ainsi, nous donnons à la fonction d'élévation `lift_slio_dlio` deux préconditions les plus faibles, `slio_wp` et `dlio_wp` : l'une de `SLIO` qui s'appelle `wp` et l'autre de `DLIO` que nous construisons à partir d'un contexte `SLIO`. Grâce à cela nous avons un lien entre la spécification statique et la spécification dynamique. Cela permet d'interconnecter les deux effets et d'avoir d'un côté un code dynamiquement vérifié et de l'autre un code vérifié statiquement. À partir de maintenant, nous pouvons écrire des programmes qui mélangent ces deux approches de vérification.

Dans le passage du monde dynamique au monde statique, il s'agit d'une simple copie du contexte CFI – `G.reveal c` dans le code. Dans le retour au monde dynamique depuis le monde statique, nous devons approximer le label dynamique – illustré par `AB` $\sqsubseteq$ `ABC` dans la figure 9.1. Pour assurer que cette approximation ait toujours lieu, nous utilisons le champ `sync` du contexte. Nous ajoutons alors deux nouveaux prédicats : l'un indique que deux contextes sont

identiques en termes de synchronisation (`no_desync`) et l'autre vérifie si un contexte est synchronisé (`is_sync`).

```
unfold let no_desync c0 c1 = G.reveal c0.sync = G.reveal c1.sync
unfold let is_sync   c0    = G.reveal c0.sync = true
```

Le prédicat `no_desync` permet d'assurer que les primitives de LIO – à l'exception de `raise` et de `setSTATE_unsafe` – ne puissent modifier la synchronisation du contexte vérifié. Cette fonction de resynchronisation est une fonction spéciale qui contrairement à toutes les primitives dynamiques peut être appelée si le contexte est désynchronisé. L'objectif de cette fonction est de mettre à jour le contexte dynamique par rapport à celui calculé par la vérification statique.

```
let raise_pre c0 l = c0.cle l ∧ c0.cur ⊑ l
let raise_post c0 x c1 l = c1 == {cur=l; cle=c0.cle; sync=G.hide true}
    ∧ monotonicity c0 c1 ∧ is_sync c1

let raise (l:labels) : DLio (unit)
    (requires λ c0 → raise_pre c0 l)
    (ensures λ c0 x c1 → raise_post c0 x c1 l)
=
    let c0 = DLIO?.get () in
    DLIO?.put ({c0 with sync=true});
    let c0 = getSTATE () in
    setSTATE ({cur=l; cle=c0.cle; sync=c0.sync});
    ()
```

Nous faisons remarquer que nous utilisons les accesseurs et mutateurs de l'effet pour statiquement changer l'état de la synchronisation du contexte. En effet, le champ `sync` du contexte `lioSTATE` n'est pas extrait à la compilation et les appels à `getSTATE` et à `setSTATE` doivent être synchronisés avec le contexte. La mise à jour dynamique permet d'assurer la cohérence du contexte à l'exécution.

Nous résumons la synchronisation du contexte dans le graphe de la figure 9.2. Si dans une fonction dans `DLIO` nous appelons une fonction dans `SLIO`, le contexte CFI sera toujours synchronisé (`no_desync`). Lorsque cette fonction dans `SLIO` se termine, nous revenons dans `DLIO` mais le contexte CFI est désynchronisé (`sync=false`). À ce moment aucune primitive de LIO ne peut être utilisée puisque le contexte n'est pas synchronisé (`is_sync` n'est pas satisfait). Ainsi, la fonction `raise` rendra un contexte CFI synchronisé.

Dans la figure 9.2, la fonction `raise` est symbolisée par un rectangle et les primitives de LIO sont modélisées par une ellipse dans leur effet respectif – nous avons deux implémentations de LIO en même temps : l'une statique dans `SLIO` et l'autre dynamique dans `DLIO`. L'opération d'élévation est représentée par une arête en pointillé et les appels de fonctions par une arête pleine.

Il existe alors deux versions de LIO dont les primitives de LIO sont préfixées d'un *s* pour

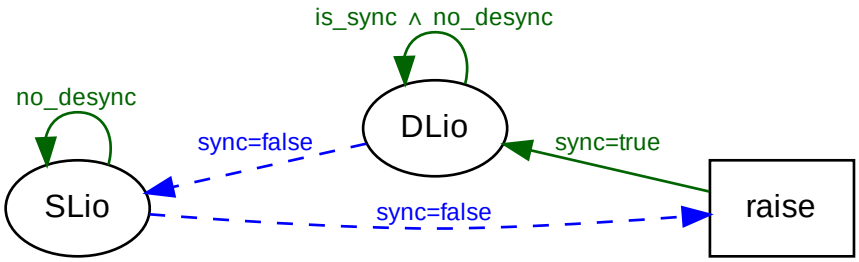

FIGURE 9.2 – Modélisation de la synchronicité entre le contexte vérifié et le contexte réel

la version statique et d'un *d* pour la version dynamique. Nous donnons à titre d'illustration la définition des valeurs labellisées du LIO statique ci-dessous :

```
noeq
type slabeled (a:Type) = {value:a; tag:G.erased labels}

let slabelOf (#a:Type) (vl:slabeled a) : Ghost (labels)
    (requires ⊤)
    (ensures λ x → x == G.reveal vl.tag)
=
    G.reveal (vl.tag)

let svalueOf (#a:Type) (vl:slabeled a) : Ghost a
    (requires ⊤)
    (ensures λ x → x == vl.value)
=
    vl.value
```

Nous faisons remarquer que le code est similaire à celui de la labellisation statique que nous avons décrite dans le chapitre 8. L'effet `Slio` est défini comme celui de la labellisation au chapitre 7. Exactement comme pour la version statique que nous avons présentée précédemment, nous utilisons les accesseurs et mutateurs de l'effet – `SLIO?.get` et `SLIO?.put` – puisque ceux-ci n'ont pas de sens à l'exécution. Nous illustrons ce changement avec la fonction `sunlabel` ci-dessous :

```
unfold let sunlabel_pre c0 vl = c0.cle (c0.cur ⊔ vl.tag)
unfold let sunlabel_post c0 x c1 vl =  x == vl.value
    ∧ c1 == {c0 with cur=(c0.cur ⊔ vl.tag);} ∧ monotonicity c0 c1
    ∧ no_desync c0 c1

let sunlabel (#a:Type) (vl:slabeled a)  : SLio (a)
```

```
    (requires λ c0 → sunlabel_pre c0 vl)
    (ensures λ c0 x c1 → sunlabel_post c0 x c1 vl)
=
    let c0 = SLIO?.get () in
    SLIO?.put ({c0 with cur=(c0.cur ⊔ slabelOf vl);});
    vl.value
```

Le code ci-dessus montre bien qu'à part la synchronisation – ici avec les prédicats `no_desync` et `is_sync` – le code de cette primitive est identique à celui de la version présentée en §8.4. Cela se reproduit aussi avec la version dynamique dont nous donnons un aperçu ci-dessous pour la primitive `dunlabel`. Cette version dynamique avec `sunlabel` est équivalente à la version de LIO en F⋆ (voir chapitre 7).

```
unfold let dunlabel_pre c0 vl = c0.cle (c0.cur ⊔ vl.tag0) ∧ is_sync c0
unfold let dunlabel_post c0 x c1 vl =  x == vl.value0
    ∧ c1 == {c0 with cur=(c0.cur ⊔ vl.tag0);} ∧ monotonicity c0 c1
    ∧ no_desync c0 c1

let dunlabel (#a:Type) (vl:dlabeled a)  : DLio (a)
    (requires λ c0 → dunlabel_pre c0 vl)
    (ensures λ c0 x c1 → dunlabel_post c0 x c1 vl)
=
    let c0 = getSTATE () in
    setSTATE ({c0 with cur=(c0.cur ⊔ vl.tag0);});
    vl.value0
```

Nous faisons remarquer que les valeurs labellisées dynamiques ont été renommées : les champs ne s'appellent plus `value` et `tag` mais `value0` et `tag0`. Cette modification est nécessaire pour aider la résolution de type de F⋆ qui, dans certains cas, échoue.

9.2 Extraction et intérêts de l'approche hybride

Nous allons maintenant illustrer l'utilisation d'une approche statique et dynamique dans un exemple extrait en langage C. Nous aurons deux fonctions : l'une fait une addition de valeurs labellisées statiquement et l'autre prend des valeurs labellisées dynamiquement et les change en version statique afin d'appeler la fonction d'addition. L'idée est de montrer comment la vérification et l'extraction sont touchées par ce changement d'approche en vérification de CFI.

Tout d'abord, l'addition statique est une fonction qui prend deux valeurs labellisées et retourne une valeur labellisée avec l'addition des deux valeurs (`svalueOf x = svalueOf a` `` `U32.add_underspec` `` `svalueOf b`) et l'union des deux labels (`slabelOf a ⊔ slabelOf b`). Nous donnons le code de cette fonction ci-dessous :

```
unfold let static_add_pre c a b = sunlabel_pre c a ∧ sunlabel_pre c b
```

```
    ∧ slabel_pre c (slabelOf a ⊔ slabelOf b)
unfold let static_add_post c0 x c1 a b = slabelOf x = (slabelOf a ⊔ slabelOf b
    )
    ∧ svalueOf x = svalueOf a `U32.add_underspec` svalueOf b
    ∧ c1 == {c0 with cur=(slabelOf a ⊔ slabelOf b)}

let static_add (a b:slabeled U32.t) : SLio (slabeled U32.t)
    (requires λ c → static_add_pre c a b)
    (ensures λ c0 x c1 → static_add_post c0 x c1 a b)
=
    let x = sunlabel a in
    let y = sunlabel b in
    slabel (x `U32.add_underspec` y) (slabelOf a ⊔ slabelOf b)
```

Dans un projet industriel, une telle fonction sera statique soit pour des raisons de performances, soit parce que sa preuve est simple. Comme nous l'avons montré dans la section 8.4, les labels disparaîtront dans le processus d'extraction.

Inversement, la fonction dynamique est moins verbeuse en termes de spécification mais beaucoup plus en termes de code. En effet, cette fonction fait l'addition de trois valeurs labellisées dynamiques en utilisant la fonction `static_add`. Ainsi, la fonction `dynamic_add` prend trois valeurs labellisées et retourne la valeur de leur addition.

```
let dynamic_add (a b c:dlabeled U32.t) : DLio (U32.t)
    (requires λ c0 → c0.cur = Nothing ∧ is_sync c0
        ∧ mcle c0 [
            _dlabelOf a;
            _dlabelOf a ⊔ _dlabelOf b;
            _dlabelOf a ⊔ _dlabelOf b ⊔ _dlabelOf c;Nothing])
    (ensures λ c0 x c1 → c0.cle == c1.cle
        ∧ c1.cur = (_dlabelOf a ⊔ _dlabelOf b ⊔ _dlabelOf c)
        ∧ x = dvalueOf c `U32.add_underspec`
            (dvalueOf a `U32.add_underspec` dvalueOf b))
=
    (* délabellisation dynamique *)
    let a' = (dunlabel a) in
    let b' = (dunlabel b) in
    let c' = (dunlabel c) in
    (* labellisation statique *)
    let a'' = (slabel a' (_dlabelOf a ⊔ _dlabelOf b ⊔ _dlabelOf c)) in
    let b'' = (slabel b' (_dlabelOf a ⊔ _dlabelOf b ⊔ _dlabelOf c)) in
    let c'' = (slabel c' (_dlabelOf a ⊔ _dlabelOf b ⊔ _dlabelOf c)) in
    (*  addition statique *)
```

```
    let x0 = static_add a'' b'' in
    let x1 = static_add c'' x0 in
    (* récupération du résultat *)
    let x' = sunlabel x1 in
    x'
```

Nous faisons remarquer que nous ne retournons pas de valeur labellisée dynamique. En fait, la fonction d'élévation doit normalement être appelée par F⋆ entre un code SLIO et un code DLIO, ce qui ne semble pas être toujours le cas. Bien que ce code ne soit pas compliqué, F⋆ a des difficultés à le valider ; cela provient probablement de la fonction d'élévation qui mélange spécification et code. Nous retournons le résultat de l'addition en l'état puis nous pouvons de nouveau utiliser des fonctions DLIO. Ci-dessous nous montrons un programme qui utilise ces deux fonctions.

```
let cle (l:labels) = true

let main (_:unit) : DLio (I32.t)
    (requires λ c0 → c0.cur = Nothing
        ∧ mcle c0 [A;B;C;AB;AC;BC;ABC;Nothing])
    (ensures λ c0 x c1 → x = 18l)
=
    raise Nothing;
    let x = (dlabel 5ul A) in
    let y = (dlabel 6ul B) in
    let z = (dlabel 7ul C) in
    let r = dynamic_add x y z in
    raise ABC;
    FStar.Int.Cast.uint32_to_int32 r
```

Nous faisons remarquer qu'après le retour de la fonction `dynamic_add`, nous continuons à utiliser des fonctions DLIO comme la fonction `raise`. Le programme retournera la valeur 18 (`x = 18l`), il s'agit de l'addition des valeurs labellisées ($5 + 6 + 7$). L'intérêt de cette partie est de montrer qu'utiliser la vérification statique en même temps que la vérification dynamique réduit considérablement la taille de la spécification. En effet, la spécification de `dynamic_add` est beaucoup plus petite que celle de `static_add`, mais cela se traduira par une augmentation du code. Cette approche hybride permet de tirer le meilleur des deux mondes en fonction de la situation. Prenons la fonction `static_add` qui est vérifiée statiquement. Cette fonction devient minimale une fois extraite en langage C.

```
uint32_t static_add(uint32_t a, uint32_t b) {
    uint32_t x = a;
    uint32_t y = b;
```

```
    return x + y;
}
```

Cependant, la fonction `dynamic_add` est beaucoup plus lourde, mais sa vérification est plus légère en termes de prédicats – dans la pratique la fonction d'élévation pose des problèmes à F⋆ qui aura besoin de beaucoup plus de ressources que prévu pour valider la fonction. Le code ci-dessous est donné à des fins d'illustration. La délabellisation que nous pouvons y observer nous montre pourquoi la version dynamique est plus coûteuse que la version statique, comme nous l'avons indiqué dans les résultats mesurés dans le chapitre 8 (voir figure 8.4).

```
uint32_t dynamic_add(   dlabeled__uint32_t a,
            dlabeled__uint32_t b,
            dlabeled__uint32_t c) {
    /* délabellisation dynamique */
    lioSTATE c00 = getSTATE((void *)(uint8_t)0U);
    lioSTATE uu____0 = c00;
    setSTATE(((lioSTATE){
        .cur = join(c00.cur, a.tag0),
        .cle = uu____0.cle
    }));
    uint32_t a_ = a.value0;
    lioSTATE c01 = getSTATE((void *)(uint8_t)0U);
    lioSTATE uu____1 = c01;
    setSTATE(((lioSTATE){
        .cur = join(c01.cur, b.tag0),
        .cle = uu____1.cle
    }));
    uint32_t b_ = b.value0;
    lioSTATE c0 = getSTATE((void *)(uint8_t)0U);
    lioSTATE uu____2 = c0;
    setSTATE(((lioSTATE){
        .cur = join(c0.cur, c.tag0),
        .cle = uu____2.cle
    }));
    uint32_t c_ = c.value0;
    /* labellisation statique */
    uint32_t a__ = a_;
    uint32_t b__ = b_;
    uint32_t c__ = c_;
    /* addition statique */
```

```
    uint32_t x = static_add(a__, b__);
    uint32_t x0 = static_add(c__, x);
    /* récupération du résultat */
    return x0;
}
```

Le programme principal ci-dessous illustre comment cette fonction est appelée. Il est tout à fait normal de commencer le programme par un appel à `raise`. En effet, le contexte n'est pas initialisé au début et cette première mise à jour nous permet d'utiliser les fonctions utilisant l'effet DLIO. Si nous avions écrit le programme entièrement dans SLIO, cette initialisation n'aurait aucun sens puisque le contexte CFI n'existerait pas. Comme elle est spécifiée et implémentée en F⋆, la fonction d'autorisation valide tous les labels.

```
bool cle(labels l) {
    return true;
}

int32_t main() {
    raise(Nothing);
    dlabeled__uint32_t x = { .value0 = (uint32_t)5U, .tag0 = A };
    dlabeled__uint32_t y = { .value0 = (uint32_t)6U, .tag0 = B };
    dlabeled__uint32_t z = { .value0 = (uint32_t)7U, .tag0 = C };
    uint32_t r = dynamic_add(x, y, z);
    raise(ABC);
    return (int32_t)r;
}
```

Dans ce chapitre, nous avons donné un exemple de combinaison entre LIO statique et LIO dynamique. Cet exemple permet de motiver l'utilisation d'une approche de CFI hybride dans un cadre d'informatique embarquée. Cependant, comment pouvons-nous assurer qu'un programme est non-interférent ?

Preuve de non-interférence

À quoi sert la propriété de non-interférence ? Nous avons vu précédemment qu'elle permet d'assurer l'absence de mélange des niveaux sensibles vers des niveaux publics. Cela nous permet d'assurer la justesse de LIO.

Dans cette partie III, nous avons montré comment implémenter LIO en F⋆ tout en tenant compte des contraintes système. Cette implémentation reprend la sémantique de LIO en tant que spécification et permet ainsi de vérifier que notre implémentation correspond à la sémantique de LIO telle qu'elle est définie dans l'état de l'art ((Stefan, Russo, Mitchell et al. 2017),(Parker, Vazou et Hicks 2019)).

Bien que cette sémantique soit prouvée comme étant non-interférente, nous examinons ici la façon de prouver cette propriété dans F⋆ directement. En effet, dans l'état de l'art il existe plusieurs approches : preuve par simulation et preuve au niveau de la sémantique du langage. Ces approches sont basées soit sur une modélisation de LIO dans un lambda-calcul ((Stefan, Russo, Mitchell et al. 2017),(Parker, Vazou et Hicks 2019)) ou comme dans la section 10.2, soit sur une preuve sur la sémantique de LIO (Algehed, Bernardy et Hritcu 2020).

Dans ce chapitre, nous montrons comment automatiser la preuve par simulation non pas sur une modélisation d'un langage, mais sur F⋆ lui-même. Cela permet d'assurer à la fois le bon comportement de la bibliothèque au niveau du programme, mais aussi d'utiliser le calcul de la précondition la plus faible pour réduire l'effort de preuve. Bien que cette preuve est incomplète à cause de facteurs de temps et des raisons techniques, cette automatisation donne une ouverture possible de l'utilisation de la métaprogrammation dans la génération automatique de preuve.

10.1 Preuve par simulation

Comment pouvoir s'assurer qu'un programme est non-interférent ? Une piste que nous avons choisie est de comparer ce programme avec une copie modifiée afin d'être non-interférente. L'approche par simulation permet d'avancer que si les deux versions du programme se comportent de la même manière alors ce programme est non-interférent. En réalité, nous pratiquons cette approche dans la vie de tous les jours pour faire un choix : nous imaginons comment ce choix impacte notre avenir puis nous comparons cet avenir avec notre présent. Si l'impact est négatif, nous renoncerons à ce choix. Cependant, si l'impact est positif alors il est probable que nous fassions ce choix.

Appliqué à la non-interférence, cela revient à "enlever" sur le programme initial les données qui pourraient interférer. Si le programme ainsi obtenu se comporte de la même manière alors

ce dernier sera non-interférent. Reprenons l'exemple de code interférent que nous avons utilisé en introduction. Une donnée sera sensible si elle est supérieure à la limite choisie : *secret* sera sensible par rapport à *public*.

```
if (secret > 0)
    print_public("ok");
else
    print_secret("ok");
```

Le programme ci-dessus est dépendant d'une information *secrète*. Ainsi, une version parfaitement non-interférente de ce programme ne pourrait pas avoir accès à l'information *secrète* si elle est considérée comme sensible puisqu'elle serait "enlevée". Grâce à cette perte d'information, nous ne pourrions pas savoir laquelle des deux fonctions – `print_public` ou `print_secret` – est appelée. Nous avons vu que pour le niveau *public*, le programme est non-interférent. Cependant, ce programme ne serait pas considéré interférent avec cette méthode pour le niveau *secret*. Cela explique pourquoi la comparaison entre les deux versions du programme doit se faire sur tous les labels.

10.1.1 Perte d'information et comparaison

Prenons un exemple de calcul pour illustrer la comparaison. Soit `f` une fonction qui prend deux valeurs labellisées et qui retourne la valeur de la première ainsi qu'une valeur labellisée qui est l'addition des deux paramètres. Cet exemple est intéressant puisqu'il illustre deux aspects de l'interférence : les flux implicites et la cohérence du CFI.

```
let f (a b:labeled int) =
    let x = unlabel a in
    let y = unlabel b in
    let m = labelOf a ⊔ labelOf b in
    let z = label (x + y) m in
    (x,z)
```

En effet, le code ci-dessus retourne deux éléments. Le premier est le résultat d'une délabellisation, ce qui est dangereux, car il s'agit d'un flux implicite. Le second élément retourné est une valeur labellisée dont la dangerosité dépendra du programme ou du CFI.

Nous illustrons une instanciation de cette fonction sur la figure 10.1. Dans cette figure, nous représentons le code de `f` instancié avec des valeurs mais nous représentons également la fonction effacée – c'est-à-dire sans toutes les valeurs sensibles – au niveau secret, que nous appelons `f'`. Nous faisons remarquer que nous représentons le niveau *secret* par `H` et le niveau *public* par `L`. Nous avons aussi choisi des valeurs qui montrent le cas où une interférence est possible.

Dans la fonction `f'` de la figure 10.1, nous effaçons la valeur de `x` mais aussi celle de `z`. Nous représentons l'effacement dans la figure par une zone noire et dans le code F⋆ par le symbole

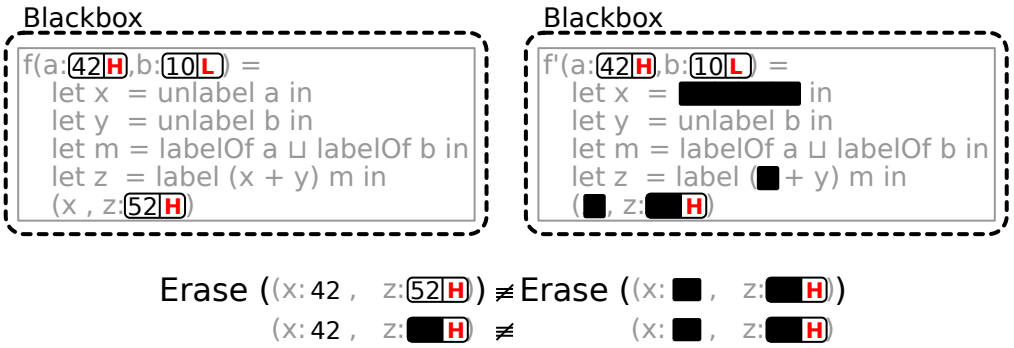

FIGURE 10.1 – Instanciation de la fonction `f` normale et effacée

" ● ". Bien qu'inutile dans cet exemple, il est important de noter que l'effacement est absorbant. Ainsi un calcul avec une information effacée retournera un résultat effacé (● `+ y =` ●). Dans cet exemple, cette règle n'a aucun impact, mais nous verrons plus tard qu'elle est nécessaire. Ainsi, la valeur labellisée `z` et la valeur qu'elle contient sont effacées.

Notons que les deux instances de la figure 10.1 ont les mêmes paramètres, mais elles ne retournent pas le même résultat. Cela vient de la variable `x` qui crée un flux implicite. En effet, cette valeur retournée peut conditionnellement avoir un impact sur le programme alors que la valeur de `x` vient du niveau *secret* (`H`). Il est bien sûr possible de corriger ce problème en labellisant le retour de la variable `x`. Le second résultat (`z`) n'est pas interférent puisqu'il est identique sur les deux versions grâce à l'utilisation correcte du CFI. En fait, si nous labellisions la variable `z` à un niveau de sensibilité inférieur à l'union des labels des paramètres (`labelOf a ⊔ labelOf b`) comme le niveau *public* (`L`), cela aurait donné un résultat différent entre les deux versions de `f`.

Bien sûr, il existe des instances où les paramètres ne permettent pas de détecter une interférence. Si nous utilisons deux paramètres qui sont *publics* (`L`) le programme sera considéré comme non-interférent. C'est en partie pour cela que nous devons comparer l'exécution sur tous les labels possibles. C'est également la raison pour laquelle la définition de l'article (SABELFELD et MYERS 2003) – équation (1) que nous rappelons ci-dessous – de la non-interférence porte sur tous les labels l possibles.

$$\forall s_1, s_2, l.\ \ s_1 =_l s_2 \Longrightarrow [\![C]\!]s_1 \approx_l [\![C]\!]s_2$$

Nous faisons remarquer que la définition ci-dessus parle aussi d'états initiaux différents mais équivalents du point de vue du label l. Ces équivalences notées $=_l$ et $\approx_l$ sont identiques à l'opération d'effacement que nous avons illustrée sur la figure 10.1.

10.1.2 Définition de la non-interférence

Notre approche est différente puisque nous n'observons pas les entrées/sorties, mais la différence entre un programme et sa version effacée, comme définie dans les travaux suivants ((LI et ZDANCEWIC 2010), (RUSSO, CLAESSEN et HUGHES 2008), (STEFAN, RUSSO, MITCHELL et al.

2017)). Ainsi, nous partons d'un même état initial x qui correspond aux paramètres de la fonction et au contexte si la fonction a un effet. Nous comparons alors le résultat de la fonction f – ce résultat peut contenir un contexte si la fonction a un effet – avec sa version effacée $f_{\mathcal{E}_l}$. Si les résultats sont équivalents au niveau l – c'est-à-dire que $(f\,x) \approx_l f_{\mathcal{E}_l}\,x$ – nous pouvons dire qu'ils sont exactement identiques après un effacement au niveau l : $\mathcal{E}_l\,(f\,x) = \mathcal{E}_l\,(f_{\mathcal{E}_l}\,x)$. Par définition, la version effacée retourne toujours des données effacées puisque toutes les valeurs labellisées y sont effacées. De plus, grâce à l'idempotence de l'effacement sur les valeurs labellisées ($\mathcal{E}_l\,(\mathcal{E}_l\,x) = \mathcal{E}_l\,x$), nous pouvons définir la non-interférence de la manière suivante :

$$\forall x, l.\ \ \mathcal{E}_l\,(f\,x) \ = \ f_{\mathcal{E}_l}\,x \tag{10.1}$$

En combinant le calcul de la précondition la plus faible de F⋆ avec la réécriture de programme avec Meta⋆, il nous apparaît possible d'automatiser la preuve en effaçant les termes labellisés puis en comparant leur précondition la plus faible. En effet, si la fonction $f_{\mathcal{E}_l}$ est non-interférente alors les propriétés sur son état initial et son état final doivent être satisfaites comme celles de la fonction f. De plus, il nous est possible de raisonner sur tous les états possibles en F⋆, ce qui permet d'appliquer la non-interférence pour tous paramètres et tous niveaux de sensibilité. Nous allons aborder l'implémentation d'une telle automatisation de preuve après la section sur la formalisation.

10.2 Illustration de la preuve originale

Comment s'assurer que la propriété de non-interférence est correcte ? Nous allons en proposer une preuve simplifiée de LIO pour tout programme afin d'illustrer la preuve présentée dans les articles (Stefan, Russo, Mitchell et al. 2017) et (Parker, Vazou et Hicks 2019). Dans cette section, nous présentons une illustration de fonctionnement de cette preuve. Nous ajoutons que les travaux cités possèdent une preuve pour un langage complet. De plus, cette preuve n'est pas complète, car elle aurait nécessité plus de temps et que dans nos travaux, nous utilisons une approche différente : l'approche utilisée dans ces articles cités fonctionne pour tout programme utilisant LIO. Cependant, notre approche décrite en section 10.3 fonctionne pour n'importe quel programme donné.

Pour montrer la justesse de cette propriété pour tout programme, nous présentons un langage simple : un lambda-calcul minimal qui ne manipule que des valeurs booléennes et des labels. L'idée est de montrer comment exprimer formellement la preuve de la non-interférence telle que définie à l'équation 10.1. Dans cette preuve, nous avons des programmes dits "effacés", c'est-à-dire des programmes où toutes les informations sensibles sont supprimées, et nous avons un effacement d'exécution où le programme n'est pas modifié, mais son observation l'est. Nous proposons une preuve qui montre que si l'exécution du programme effacé est indistinguable de l'effacement de l'exécution du programme à un niveau l, le programme est alors non interférent

au niveau l.

Définition du lambda-calcul

Notre version minimale de LIO est une version qui utilise uniquement des entiers naturels comme label et peut également utiliser des booléens comme valeurs. Par convention, les noms de variables en majuscules concernent des valeurs concrètes et le nom de variable en minuscules concerne des termes non réduits.

Ce langage définit des termes qui permettent de structurer le langage. Ces termes peuvent être des entiers naturels pour les labels ($\mathbb{N}$), des opérations entre labels $\oplus$, des variables, des valeurs labellisées $\langle\, t,\, \mathbb{N} \,\rangle$ (ainsi qu'une opération d'ouverture unlabel $\langle\, t,\, \mathbb{N} \,\rangle$ et de création label $t\, \mathbb{N}$ des valeurs labellisées), des applications ou des lambdas. Ci-dessous sont donnés les termes définis par le langage :

$$t, u, v, w := \mathbb{N} \mid x \mid \langle\, t,\, u \,\rangle \mid t \oplus u \mid (t\, u) \mid (\lambda\, x.\, e) \mid \text{label}\, t\, u \mid \text{unlabel}\, t \tag{10.2}$$

Les opérations sur les labels ($\oplus$) concernent les trois opérations sur les labels décrit dans le chapitre 7 : $\sqsubseteq$, $\sqcap$ et $\sqcup$.

$$\oplus := \sqsubseteq \mid \sqcap \mid \sqcup \tag{10.3}$$

Les valeurs peuvent être des labels $\mathbb{N}$, des lambdas $(\lambda\, x.\, e)$, des valeurs labellisées $\langle\, V,\, L \,\rangle$ (couple entre une valeur et un label) et des booléens à savoir true ou false.

$$L, V, W, R := \mathbb{N} \mid \langle\, V,\, L \,\rangle \mid (\lambda\, x.\, e) \mid \text{true} \mid \text{false} \tag{10.4}$$

Dans ce formalisme, nous utilisons une convention entre termes réduits (majuscules) et termes non réduits (minuscules). Par exemple, un terme qui se réduit vers un label pourra être noté M ou N. Parfois, certains labels seront notés l avec un indice. Cela concerne essentiellement le label courant.

$$M, N \in \mathbb{N}$$
$$l, l_n \in \mathbb{N} \text{ et } n \in \{0, 1, 2, 3, 4, \ldots\}$$

Les opérations de combinaison des labels ($\sqcap$ et $\sqcup$) prennent deux labels et en retournent un nouveau ($\mathbb{N} \to \mathbb{N} \to \mathbb{N}$). L'opération de comparaison de labels prend deux labels et retourne une valeur booléenne ($\mathbb{N} \to \mathbb{N} \to \{\text{true}, \text{false}\}$). Nous notons l'évaluation de ces opérateurs$[\![\oplus]\!]$.

$$\begin{aligned} [\![l_0 \sqsubseteq l_1]\!] &= l_0 \leq l_1 \\ [\![l_0 \sqcap l_1]\!] &= \min l_0\, l_1 \\ [\![l_0 \sqcup l_1]\!] &= \max l_0\, l_1 \end{aligned} \tag{10.5}$$

À ce stade, nous avons un lambda calcul qui nous permet d'exprimer des programmes. En

donnant une sémantique, nous décrivons comment typer le programme. Ci-dessous nous décrivons une sémantique à grands pas pour montrer ce qu'est un programme correctement typé.

$$\text{App}\ \frac{l_1 \vdash t \Downarrow l_2 \vdash (\lambda\, x.\, R) \qquad l_2 \vdash u \Downarrow l_3 \vdash v \qquad l_3 \vdash R[V/x] \Downarrow l_3 \vdash W}{l_1 \vdash (t\, u) \Downarrow l_3 \vdash W}$$

$$\text{LabelOps}\ \frac{l_1 \vdash t \Downarrow l_2 \vdash N \qquad l_2 \vdash u \Downarrow l_3 \vdash M}{l_1 \vdash t \oplus u \Downarrow l_3 \vdash [\![N \oplus M]\!]}$$

$$\text{Label}\ \frac{l_1 \vdash t \Downarrow l_2 \vdash T \qquad l_2 \vdash u \Downarrow l_3 \vdash l \qquad l_3 \leq l}{l_1 \vdash \text{label}\, t\, u \Downarrow l_3 \vdash \langle\, T,\, l\,\rangle}$$

$$\text{Unlabel}\ \frac{l_1 \vdash t \Downarrow l_2 \vdash \langle\, V,\, l\,\rangle}{l_1 \vdash \text{unlabel}\, t \Downarrow l_2 \vdash V}$$

$$\text{Valeurs}\ \frac{R \in \{\mathbb{N}, \langle\, V,\, l\,\rangle, (\lambda\, x.\, e)\}}{l_1 \vdash R \Downarrow l_1 \vdash R}$$

Comme expliqué en début de chapitre, prouver la non-interférence nécessite de "perdre" de l'information. Dans notre formalisme, cette perte d'information est notée $\bullet$. Ci-dessous, nous rajoutons aux valeurs possibles 10.4, les valeurs effacées :

$$L, V, W, R := \mathbb{N} \mid \langle\, V,\, L\,\rangle \mid (\lambda\, x.\, e) \mid \text{true} \mid \text{false} \mid \bullet \tag{10.6}$$

Nous définissons un terme supplémentaire $\varepsilon_l\, t$ qui efface un terme t si le label courant est inférieur au label l. La sémantique de ce nouveau terme est donnée ci-dessous. Toutefois, nous faisons remarquer qu'à l'instar de la valeur effacée, ce terme est utilisé uniquement pour la preuve et n'a pas de sens à l'exécution.

$$\text{ValeursEff}\ \frac{R \neq \langle\, V,\, l\,\rangle}{l_1 \vdash \varepsilon_l\, R \Downarrow l_1 \vdash R}$$

$$\text{ValeursLabEff}\ \frac{l_1 > M}{l_1 \vdash \varepsilon_l\, \langle\, V,\, M\,\rangle \Downarrow l_1 \vdash \langle\, \bullet,\, M\,\rangle}$$

$$\text{ValeursLab}\ \frac{l_1 \leq M}{l_1 \vdash \varepsilon_l\, \langle\, V,\, M\,\rangle \Downarrow l_1 \vdash \langle\, V,\, M\,\rangle}$$

Les valeurs effacées sont absorbantes, c'est-à-dire que l'effacement est propagé au résultat. Un programme qui respecte l'effacement doit respecter cette absorbance. La sémantique de l'effacement est donnée ci-dessous :

$$\text{AppEffGauche}\ \frac{}{l_1 \vdash (\bullet\, (\varepsilon_l\, u)) \Downarrow l_1 \vdash \bullet}$$

$$\text{AppEffDroite}\ \frac{}{l_1 \vdash ((\varepsilon_l\, t)\, \bullet) \Downarrow l_1 \vdash \bullet}$$

$$\text{OpsEffGauche}\ \frac{}{l_1 \vdash \bullet \oplus (\varepsilon_l\, u) \Downarrow l_1 \vdash \bullet}$$

$$\text{OpsEffDroite}\ \frac{}{l_1 \vdash (\varepsilon_l\, t) \oplus \bullet \Downarrow l_1 \vdash \bullet}$$

Une fois que l'effacement est décrit, nous précisons comment cet effacement se propage inductivement dans les termes. En effet, effacer un terme revient à effacer tous ses sous-termes. Ainsi, nous donnons la sémantique de cet effacement.

$$\text{LabEff}\ \frac{}{l_1 \vdash \varepsilon_l\, (\mathsf{label}\, t\, u) \Downarrow l_1 \vdash \mathsf{label}\, (\varepsilon_l\, t)\, (\varepsilon_l\, u)}$$

$$\text{UnlabEff}\ \frac{}{l_1 \vdash \varepsilon_l\, (\mathsf{unlabel}\, t) \Downarrow l_1 \vdash \mathsf{unlabel}\, (\varepsilon_l\, t)}$$

$$\text{OpsEff}\ \frac{}{l_1 \vdash \varepsilon_l\, (t \oplus u) \Downarrow l_1 \vdash (\varepsilon_l\, t) \oplus (\varepsilon_l\, u)}$$

$$\text{AppEff}\ \frac{}{l_1 \vdash \varepsilon_l\, (t\, u) \Downarrow l_1 \vdash ((\varepsilon_l\, t)\, (\varepsilon_l\, u))}$$

Maintenant, nous avons un lambda calcul qui nous permet d'écrire des programmes et qui définit ce qu'est un programme bien typé. La preuve de non-interférence consiste à montrer que pour tout label un attaquant qui observe à un label donné, ne peut pas observer des informations protégées par un label supérieur.

Pour prouver cette propriété, nous pouvons montrer que "*Pour tout terme et label, l'observabilité entre une exécution effacée et un effacement de l'exécution est indistinguable.*" Cela revient à comparer deux exécutions : l'une où l'effacement intervient après l'exécution et l'autre où l'effacement intervient avant. En utilisant notre formalisme, une telle propriété s'exprime de la manière suivante :

$$P(t,l) : \forall l_1.\ (l_1 \vdash t \longrightarrow l_2 \vdash u \wedge l_1 \vdash \varepsilon_l\, t \longrightarrow l_3 \vdash u') \Longrightarrow (l_2 \vdash \varepsilon_l\, u \longrightarrow l_4 \vdash \varepsilon_l\, u' \wedge u' = \varepsilon_l\, u) \tag{10.7}$$

Visuellement, nous pouvons représenter cette preuve sous forme d'un schéma : l'exécution normale du terme t doit une fois effacée être équivalent à l'exécution du terme t effacé.

$$\begin{array}{ccc} l_1 \vdash t & \longrightarrow & l_2 \vdash u \\ \Big\downarrow \varepsilon_l & & \Big\downarrow \varepsilon_l \\ l_1 \vdash \varepsilon_l\, t & \longrightarrow & l_3 \vdash \varepsilon_l\, u' \end{array}$$

En utilisant notre formalisme, nous pouvons avoir une intuition d'une preuve. Pour assurer que l'implémentation est correcte, nous raisonnons sur chaque terme qui se réduise en valeurs (cas *a* et *b*). Puis nous raisonnons sur les termes où une induction a lieu dans les autres cas. Dans ces derniers, nous supposons que la propriété de non-interférence est valide pour les sous-termes et nous montrons que le terme courant préserve la propriété.

a. Si $l_1 \vdash t \longrightarrow l_2 \vdash \langle V, L \rangle$, nous avons deux cas :
   — Le terme est "observable" $L \leq l$:
   $l_1 \vdash t \longrightarrow l_2 \vdash \langle V, L \rangle$ alors $l_1 \vdash \varepsilon_l\, t \longrightarrow l_3 \vdash \varepsilon_l \langle V, L \rangle$ et comme $l_2 \vdash \varepsilon_l (\langle V, L \rangle) \longrightarrow l_2 \vdash \langle V, L \rangle = l_3 \vdash \varepsilon_l (\langle V, L \rangle) \longrightarrow l_3 \vdash \langle V, L \rangle$, la proposition P est satisfaite.
   — Le terme n'est pas "observable" $L > l$:
   $l_1 \vdash t \longrightarrow l_2 \vdash \langle V, L \rangle$ alors $l_1 \vdash \varepsilon_l\, t \longrightarrow l_3 \vdash \varepsilon_l \langle V, L \rangle$ et comme $l_2 \vdash \varepsilon_l (\langle V, L \rangle) \longrightarrow l_2 \vdash \langle \bullet, L \rangle = l_3 \vdash \varepsilon_l (\langle V, L \rangle) \longrightarrow l_3 \vdash \langle \bullet, L \rangle$, la proposition P est satisfaite.

b. Si $l_1 \vdash t \longrightarrow l_2 \vdash V$ avec $V \in \{\mathbb{N}, \text{true}, \text{false}, (\lambda\, x.\, e)\}$ alors $l_1 \vdash \varepsilon_l\, t \longrightarrow l_3 \vdash \varepsilon_l\, V$ et comme $l_2 \vdash \varepsilon_l\, V \longrightarrow l_2 \vdash V$ et $l_3 \vdash \varepsilon_l\, V \longrightarrow l_3 \vdash V$ alors la proposition P est satisfaite.

c. Par hypothèse d'induction sur t et sur u, on a $l_1 \vdash v \longrightarrow l_2 \vdash w$ et $l_1 \vdash \varepsilon_l\, v \longrightarrow l_3 \vdash \varepsilon_l\, w'$ et $\varepsilon_l\, w' = \varepsilon_l\, w$. Si $l_1 \vdash t \oplus u \longrightarrow l_2 \vdash [\![N \oplus M]\!]$ et $l_1 \vdash \varepsilon_l\, (t \oplus u) \longrightarrow l_1 \vdash (\varepsilon_l\, t) \oplus (\varepsilon_l\, u)$ alors on a $N' = \varepsilon_l\, t$ et $M' = \varepsilon_l\, u$ et $l_1 \vdash [\![N' \oplus M']\!] = l_2 \vdash [\![N \oplus M]\!]$

d. Si $l_1 \vdash \text{label}\, t\, u \longrightarrow l_2 \vdash \langle V, L \rangle$ alors $l_1 \vdash \varepsilon_l\, (\text{label}\, t\, u) \longrightarrow l_2 \vdash \varepsilon_l\, (\langle V, L \rangle)$, or la satisfaisabilité de P dépend de $\langle V, L \rangle$.

e. Si $l_1 \vdash \text{unlabel}\, t \longrightarrow l_2 \vdash V$ alors $l_1 \vdash \varepsilon_l\, (\text{unlabel}\, t) \longrightarrow l_2 \vdash \varepsilon_l\, V$, or la satisfaisabilité de P dépend de $\langle V, L \rangle$ (La règle de Unlabel réduit t à $\langle V, L \rangle$).

f. Par hypothèse d'induction sur t et sur u, on a $l_1 \vdash v \longrightarrow l_2 \vdash w$ et $l_1 \vdash \varepsilon_l\, v \longrightarrow l_3 \vdash \varepsilon_l\, w'$ et $\varepsilon_l\, w' = \varepsilon_l\, w$. Si $l_1 \vdash (t\, u) \longrightarrow l_2 \vdash R$ et $l_1 \vdash \varepsilon_l\, (t\, u) \longrightarrow l_1 \vdash ((\varepsilon_l\, t)\, (\varepsilon_l\, u))$ alors $l_1 \vdash ((\varepsilon_l\, t)\, (\varepsilon_l\, u)) = l_2 \vdash \varepsilon_l\, R$

□

Liens avec l'implémentation

Le langage F⋆ est proche du lambda-calcul. De plus, plusieurs travaux précédents ont confirmé la justesse de la propriété de non-interférence (Stefan, Russo, Mitchell et al. 2017) et (Parker, Vazou et Hicks 2019). De fait, notre travail ne consistera pas à montrer que la non-interférence est valide, mais à automatiser cette preuve en s'appuyant sur le calcul de la précondition la plus faible de F⋆. Dans certains cas, cela permet de déterminer de manière automatique si un programme est interférent ou pas.

10.3 Automatisation de la preuve de non-interférence

Comment effacer une information ? En F⋆, le calcul de la précondition la plus faible permet de connaître le domaine de valeur d'une variable. Par exemple, une fonction identité permet de lier le résultat par rapport à l'entrée : `id : (i:nat) →Tot (o:nat{o = i})`. Ainsi faire

perdre une information c'est garantir l'absence de ce lien mais également garantir l'absence de comparaison des valeurs. En effet, l'idée est de conserver le moins d'information possible, mais de perdre le raffinement qui décrit un ensemble précis de valeurs. Nous définissons en F⋆ une fonction – appelée ● – qui retourne une valeur quelconque pour un type donné.

```
assume val ● : (#a:Type) → a
```

La fonction ci-dessus n'a pas de code, puisqu'une implémentation n'a pas de sens : comment choisir cette valeur quelconque ? En réalité, nous n'avons pas à choisir une valeur quelconque, car ce qui importe c'est d'avoir une précondition la plus faible qui indique que la valeur se situe dans le domaine de valeur du type. Ainsi, cette fonction n'a pas à être extraite, puisqu'elle ne sert qu'à la vérification. Par exemple, l'objectif de la fonction ● `#int` est de retourner une valeur non fixée arbitrairement. Ainsi, λ `()` →`3` et λ `()` →● `#int` ont respectivement la précondition la plus faible λ `x` →`x = 3` et la précondition la plus faible λ `x` →⊤ – `x` est le retour de la fonction – et par conséquent sont inégales. La première retourne une constante et la seconde retourne une valeur qui peut être égale ou non à une constante donnée. Nous pouvons vérifier les propriétés suivantes en F⋆ :

```
let ●_IsReflexive =
    assert(∀ (a:Type). ● #a == ● #a)

let ●_ValueIsUnknown =
    assert(∀ (a:Type) (x:a). (● #a == x) ∨ (● #a =!= x))
```

La première propriété indique l'identité entre deux valeurs effacées. La deuxième propriété précise qu'il est impossible de connaître la valeur effacée pour n'importe quel type – elle est soit égale soit différente. Ces deux propriétés montrent que nous pouvons comparer uniquement l'état des valeurs : nous pouvons savoir si elles sont effacées ou pas. Nous faisons remarquer que le type de l'effacement est parfois précisé pour aider l'inférence de type de F⋆ – par exemple `(● #int) = 3` précise que l'effacement est de type `int`.

Nous avons indiqué que l'effacement est absorbant. C'est-à-dire que tout calcul avec une valeur effacée (●) donne un résultat effacé. Dans F⋆, l'opération ● `+ y:nat{y ≥ 3}` conserve le raffinement (`nat + nat:{y ≥ 3}`). Toutefois dans ce cas, nous ne souhaitons pas garder le raffinement. Imaginons le code `addition_problem` suivant :

```
let addition_problem =
    let f (y:nat{y ≥ 3}) = ● + y in
    assert(∀ y. f y = ●)
```

Le code ci-dessus n'est pas accepté par F⋆. En effet, ● ≠ `(● + y:nat{y ≥ 3})`. Nous devons ajouter une règle spécifique d'absorption. En réalité, nous devons ajouter ces règles pour tous les types primitifs et leurs opérateurs : les entiers, les booléens, etc. Nous donnons ces axiomes sur les entiers (`int`) pour l'addition, la soustraction et l'opérateur inférieur. De fait, ces axiomes marchent également sur les entiers naturels comme les (`nat`).

```
let axiom_plusAbsorbanceL n : Lemma (● + n = ●)
    [SMTPat (● + n)] = admit ()
let axiom_plusAbsorbanceR n : Lemma (n + ● = ●)
    [SMTPat (n + ●)] = admit ()

let axiom_minusAbsorbance () : Lemma (- ● = ●)
    [SMTPat (- (● #int))] = admit ()

let lemma_minusAbsorbanceL n : Lemma (n - ● = ●)
    [SMTPat (n - ●)]
=
    axiom_plusAbsorbanceR n

let lemma_minusAbsorbanceR n: Lemma (● - n = ●)
    [SMTPat (● - n)]
=
    axiom_plusAbsorbanceL n

let axiom_lessthanL v : Lemma (● < v = ●)
    [SMTPat (● < v)] = admit ()
let axiom_lessthanR v : Lemma (v < ● = ●)
    [SMTPat (v < ●)] = admit ()
```

Nous faisons remarquer que nous utilisons l'attribut `SMTPat` qui indique sur quelle forme du code – ici `● + n` par exemple – appliquer automatiquement le lemme. Cela permet de valider le code `addition_problem`, car la fonction ● est absorbante.

Maintenant, nous pouvons écrire la fonction d'effacement. Cette fonction transforme toute valeur labellisée en valeur labellisée effacée si et seulement si celle-ci est au-dessus d'un certain label. Cette fonction n'a pas besoin d'autres règles puisqu'elle ne doit pas avoir d'impact sur le contexte.

```
let ℰ_l (#a:eqtype) (vl:labeled a) : labeled a =
    if labelOf vl ⊑ l then
        vl
    else
        {value=(● #a); tag=(labelOf vl)}
```

Comment nous assurer que cette fonction est correcte ? Nous avons deux cas à prendre en compte par rapport au label d'effacement : soit le label de la valeur labellisée est trop haut – *secret* par exemple – et dans ce cas la valeur labellisée doit être effacée (●), soit le label est égal ou plus faible – *public* par exemple – et dans cet autre cas la valeur labellisée doit être identique et comparable. Comme dans l'article (Stefan, Russo, Mitchell et al. 2017), nous vérifions ces deux cas de la manière suivante :

```
let secretAreDot  =
    assert(∀ (a:eqtype) (l:labels) (vl:labeled a{¬(labelOf vl ⊑ l)}).
        (ℰ_l vl).value = ●)

let publicAreSame =
    assert(∀ (a:eqtype) (l:labels) (vl:labeled a{labelOf vl ⊑ l}).
        (ℰ_l vl).value = vl.value)
```

Notons également que la fonction $\mathcal{E}_l$ est idempotente ; c'est-à-dire que quel que soit le nombre de fois qu'elle est appliquée, elle aura le même effet. Dans notre cas, effacer de multiples fois une valeur labellisée aura le même résultat que l'effacer une seule fois. Cette propriété est également prise en compte dans l'article (Stefan, Russo, Mitchell et al. 2017), ce que nous vérifions aussi en F⋆ :

```
let idempotentOverLabeledValues =
    assert(∀ (a:eqtype) (vl:labeled a) l.
        ℰ_l vl = ℰ_l (ℰ_l vl))
```

Nous pouvons dire de manière simplifiée que pour vérifier la non-interférence, il nous suffit de remplacer toutes les valeurs labellisées d'un programme par un appel à fonction $\mathcal{E}_l$. Nous allons donc montrer comment utiliser cette fonction dans F⋆. Toutefois, pour écrire des programmes interférents, nous ne pouvons pas utiliser LIO. En effet, la spécification que nous avons définie en F⋆ nous empêche d'écrire un programme qui soit interférent. C'est pourquoi nous illustrons cette automatisation dans un CFI qui est volontairement incorrect afin de montrer plus facilement comment cette automatisation de preuve montre l'absence ou la présence d'interférence.

10.3.1 *El LIO*

Dans ce paragraphe nous présentons *El LIO* – signifie littéralement "le bazar" en espagnol –, une version de LIO conçue pour permettre une liberté totale de développement : elle n'impose aucune contrainte. Cette version d'exemple n'a pas de contexte de CFI et n'a aucune pré- et post-condition sur les primitives CFI. Il est évident qu'une telle implémentation est totalement dangereuse et reste uniquement pédagogique.

Treillis utilisé

Nous utilisons un treillis sur les nombres naturels avec pour lois internes les fonctions maximum et minimum. L'opération de comparaison $\sqsubseteq$ est strictement identique au test inférieur ou égal $\leq$. Ce treillis a été choisi, car il est simple à prouver. En effet, l'ensemble des lemmes sur les entiers naturels de F⋆ permettent de valider automatiquement les invariants du treillis. Nous avons précédemment montré comment écrire un treillis plus réaliste.

```
type labels = nat

unfold let ⊔ (a b:labels) : labels  = if a < b then b else a
unfold let ⊓ (a b:labels) : labels  = if a < b then a else b
unfold let ⊑ (a b:labels) : bool = a ≤ b
```

Bien que cette version de CFI soit totalement fausse, le treillis que nous utilisons est correct. Nous donnons la preuve qui est automatisée de la même manière que celle que nous avons décrite au chapitre 7. Dans notre cas, F⋆ prouvera automatiquement ce treillis comme montré ci-dessous :

```
let latticeIsCorrect : lattice labels = {
    ⊑ = ⊑;
    ⊔ = ⊔;
    ⊓ = ⊓;
    bottom = 0;
    lawFlowReflexivity  = (λ _ → ());
    lawFlowAntisymetry  = (λ _ _ → ());
    lawFlowTransitivity = (λ _ _ _ → ());
    lawJoin = (λ _ _ _ →  ());
    lawMeet = (λ _ _ _ →  ());
}
```

En effet, si notre définition de treillis n'était pas correcte, F⋆ ne pourrait pas instancier le type `lattice` sans que l'utilisateur apporte une preuve comme argument. Nous faisons également remarquer que ce treillis n'est pas extractible puisque le type `nat` est non-borné comme le type `int`.

Primitives de CFI

Dans cette partie, nous décrivons une structure pour manipuler des données labellisées : une fonction de labellisation, une fonction de délabellisation et une fonction pour inspecter le label d'une valeur labellisée. Nous donnons le code ci-dessous :

```
type labeled (a:eqtype) = {v:a; t:labels}

let mkLabeled (#a:eqtype) (v:a) (l:labels) = {v=v; t=l}
let unLabeled (#a:eqtype) (vl:labeled a) = vl.v
let labelOf (#a:eqtype) (vl:labeled a) = vl.t

let ℰ_l (#a:eqtype) (vl:labeled a): labeled a =
    if labelOf vl ⊑ l then
        vl
    else
```

```
    mkLabeled (● #a) (labelOf vl)
```

Nous rappelons que sans contexte ni pré- et post-condition appropriés, il est certain que cette implémentation accepte des programmes interférents. C'est cela que nous montrons dans le paragraphe suivant.

10.3.2 Vérification manuelle

Dans cette partie, nous allons vérifier la non-interférence en créant nous-même la version effacée. Nous allons commencer par un programme non-interférent : la relabellisation de valeurs. Ce programme délabellise puis relabellise une valeur labellisée. Nous donnons deux versions ci-dessous : `goodRelabelling` qui est non-interférent et `badRelabelling` qui au contraire est interférent.

```
let goodRelabelling (vx:labeled int) : labeled int =
    let x = unLabeled vx in
    mkLabeled x (labelOf vx)

let badRelabelling (vx:labeled int) : labeled int =
    let x = unLabeled vx in
    mkLabeled x 0
```

La différence entre ces fonctions est la monotonie du label dans la valeur labellisée. En effet, le label est monotone dans la fonction `goodRelabelling` alors que dans la fonction appelée `badRelabelling`, il est remis au minimum c'est-à-dire `0`. Cette dernière fonction est interférente pour toutes les valeurs labellisées dont le label est supérieur à 0. Dans LIO c'est le label courant qui assure la monotonie des labels même pour les valeurs labellisées. Pour vérifier ces fonctions nous implémentons leur version effacée :

```
let goodRelabelling_{E_l} (vx:labeled int) : labeled int =
    let x = unLabeled (E_l vx) in
    E_l (mkLabeled x (labelOf vx))

let badRelabelling_{E_l} (vx:labeled int) : labeled int =
    let x = unLabeled (E_l vx) in
    E_l (mkLabeled x 0)
```

Enfin pour vérifier la non-interférence des fonctions `goodRelabelling` et `badRelabelling`, nous écrivons notre définition de la non-interférence sous forme d'assertions. Nous précisons que l'attribut `[@expect_failure]` ne valide le code que si ce dernier échoue à la vérification.

```
let goodRelabellingIsNI =
    assert(∀ vx l. E_l (goodRelabelling vx) = goodRelabelling_{E_l} vx)

[@expect_failure]
```

```
let badRelabellingIsNotNI =
    assert(∀ vx l. ℰ_l (badRelabelling vx) = badRelabelling_ℰ_l vx)
```

Ainsi, nous voyons que la fonction `badRelabelling` qui est interférente est bien refusée, contrairement à la fonction `goodRelabelling` qui ne l'est pas. Prenons un autre exemple, comme le programme `f` illustré sur la figure 10.1. Ce programme retourne un flux implicite et une valeur labellisée. Nous avons précédemment expliqué en quoi ce programme est interférent. Nous remplaçons toutes les occurrences de valeur labellisée par un appel à la fonction $\mathcal{E}_l$.

```
let f_ℰ_l (a b:labeled int) =
    let x = unLabeled (ℰ_l a) in
    let y = unLabeled (ℰ_l b) in
    let m = labelOf (ℰ_l a) ⊔ labelOf (ℰ_l b) in
    let z = ℰ_l (mkLabeled (x + y) m) in
    (x, ℰ_l z)
```

Maintenant, nous avons deux fonctions : `f` et $\texttt{f}_{\mathcal{E}_l}$. Ces fonctions peuvent être utilisées pour prouver la non-interférence de la manière qui suit :

```
[@expect_failure]
let allResultsAreNI =
    assert(∀ va vb l.
        (let (x0,z0) = f va vb in
         let (x1,z1) = f_ℰ_l va vb in
         (ℰ_l z0) = z1 ∧ x1 = x0))
```

Sans surprise, le programme échoue puisque les retours `x0` et `x1` ne correspondent pas pour certains labels. En fait, la variable `x1` peut être égale à ● puisqu'il existe des cas où la valeur labellisée `va` sera effacée : ce flux implicite peut être interférent. En ne vérifiant la non-interférence que sur le retour labellisé, nous pouvons souligner la source d'interférence dans cette fonction :

```
let onlyLabeledIsNI =
    assert(∀ va vb l.
        (let (x0,z0) = f va vb in
         let (x1,z1) = f_ℰ_l va vb in
         (ℰ_l z0) = z1))
```

Nous rappelons que `z1` est déjà effacé. Nous avons vu qu'il est possible de prouver la non-interférence en F⋆. Cependant, le processus de réécriture est très répétitif et se prête totalement à une automatisation. Nous montrons dans la partie suivante comment utiliser Meta⋆ pour réécrire les programmes F⋆ afin de jouer cette preuve automatiquement.

10.3.3 Vérification automatique

Dans cette partie, nous abordons l'automatisation de cette preuve par effacement. L'algorithme de transformation va "modifier" l'Abstract Syntax Tree (AST) – la représentation du

programme – en y ajoutant à chaque valeur labellisée des appels à $\mathcal{E}_l$. Nous donnons une représentation graphique sur la figure 10.2 avec l'AST et les appels à $\mathcal{E}_l$ ajoutés par la fonction de "modification".

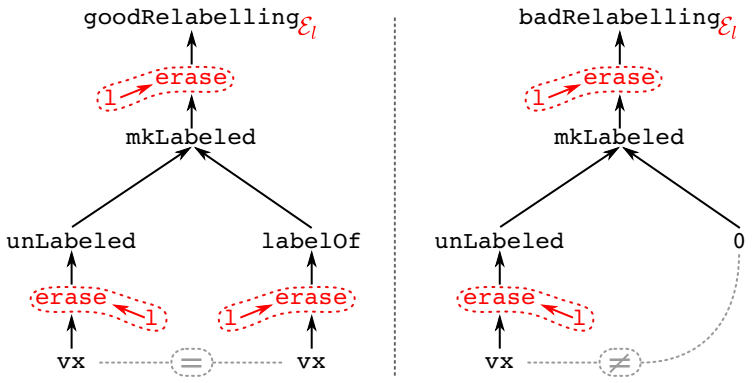

FIGURE 10.2 – Les ASTs et les ajouts de la fonction de "modification".

Cette fonction de modification ouvre les termes inductivement et parcourt l'ensemble des éléments en retournant un nouvel AST où les dépendances sont également effacées. Nous illustrons la nécessité de parcourir l'intégralité du code avec le code suivant :

```
let vhigh (z:int) = mkLabeled z 10

let call_vhigh (vx:labeled int) : labeled int =
    let x = unLabeled vx in
    let y = unLabeled (vhigh 5) in
    mkLabeled (x + y) (labelOf vx ⊔ labelOf (vhigh 5))
```

Si nous devions appliquer notre fonction d'effacement, nous serions obligés de l'appliquer aussi à la fonction `vhigh`. Dans le cas contraire, si les fonctions qui sont appelées sont interférentes, cela rendrait notre programme interférent tout en validant la preuve. Nous devons donc "modifier" l'intégralité de la fonction `call_vhigh`. Pour cela, notre fonction de "modification" va simplement créer un nouvel AST complètement différent du premier. La fonction `generate_patched_defs` prend trois termes : la fonction d'effacement, le type des labels et la fonction à transformer (`entrypoint`).

```
let generate_patched_defs (blacklist: list (bool * name))
    (patchFunction globalArgType: term)
    (entrypoint: term)
    : Tac (list sigelt)
=
    let defs = patch_term_and_defs
        blacklist
        globalArgType
        patchFunction
```

```
        entrypoint
    in
    make_sglets (defs)
```

La fonction `generate_patched_defs` retourne une liste de noms qui peuvent avoir un type et une définition. Il s'agit du nouveau terme ainsi que toutes ses dépendances "modifiées". Puis la fonction `make_sglets` transforme ce nouveau terme en une liste de `sigelt` qui sont des termes *top-level* – ces termes correspondent aux définitions du module. Nous ajoutons le suffixe `_patched` aux noms de ces nouveaux termes afin d'éviter une collision avec les anciens qui existent toujours. À ce moment nous venons d'enrichir le module avec de nouveaux termes effacés. Les fonctions `patch_term_and_defs` et `make_sglets` étant complexes et dépendantes de Meta⋆, nous n'allons pas détailler le fonctionnement interne.

Nous notons que les primitives LIO – ou de n'importe quel autre CFI – ne doivent pas être effacées. Nous donnons alors une liste de fonctions à ne pas effacer. Dans le cas de *El LIO*, nous donnons la fonction de dé-labellisation, la fonction de labellisation et le type des labels.

```
let mod = "Module_name"
let bl = [
    true, [mod; "unLabeled"];
    true, [mod; "mkLabeled"];
    true, [mod; "labeled"];
    true, [mod; "labels"]
]
```

Pour générer le nouveau terme, nous appelons la fonction `generate_patched_defs` avec la liste de définitions à ne pas effacer. Nous donnons la fonction d'effacement `eraseE` qui est juste un appel à la fonction $\mathcal{E}_l$ mais avec une signature plus explicite – le type est explicitement donné contrairement à $\mathcal{E}_l$. Nous donnons le type des labels – ici `nat` – ainsi que la fonction à effacer. Tous ces arguments sont transformés en termes Meta⋆ via l'opérateur " ` ".

Le résultat que nous obtenons est une liste de `sigelt`. Nous les passons à la fonction `%splice` qui injecte les définitions dans le module. Cela nous permet d'utiliser les fonctions générées. Ainsi dans le code ci-dessous, nous rejouons les mêmes preuves que dans la partie précédente, mais cette fois nous n'avons pas implémenté la version effacée de la fonction.

```
%splice[](generate_patched_defs bl (`eraseE) (`nat) (`goodRelabelling))

let goodRelabellingIsNI_auto =
    assert(∀ vx l. ℰ_l (goodRelabelling vx) = goodRelabelling_patched l vx)

%splice[](generate_patched_defs bl (`eraseE) (`nat) (`badRelabelling))

[@expect_failure]
let badRelabellingIsNotNI_auto =
```

```
assert(∀ vx l. ℰ_l (badRelabelling vx) = badRelabelling_patched l vx)
```

Dans les vérifications que nous avons présentées, nous ne parlons pas de contexte. En fait, l'application de cette preuve dans un CFI implique d'effacer les éléments du contexte. Ainsi, dans un gestionnaire de mémoire cela revient à comparer le contexte CFI mais également la mémoire. Dans ce cas, la fonction `f` doit retourner un résultat et un contexte. En plus des résultats, les contextes doivent être équivalents au label l. À l'heure actuelle, ce résultat n'est pas techniquement opérationnel.

10.4 Limites actuelles et directions futures

Dans cette section, nous avons présenté une mécanisation de la preuve de non-interférence et une implémentation de LIO en F⋆ pouvant être extraite en langage C, vérifiée statiquement et vérifiée dynamiquement (Marty et al. 2020). À des fins pédagogiques, nous avons simplifié le code technique pour le rendre plus lisible que celui que nous avons développé. En effet, cela est compliqué à exprimer dans l'état actuel de F⋆.

De plus, nous avons beaucoup avancé dans la réalisation de LIO⋆ – grâce à Lucas Franceschino – puisque nous utilisons les *layered effects*, une fonctionnalité qui permet la combinaison des effets. Cette fonctionnalité n'était pas utilisable à cause de problèmes d'extraction qui sont – au moment de l'écriture de ces lignes – en cours de correction. Contrairement à la fonction d'élévation que nous présentons dans le chapitre 9, notre bibliothèque LIO⋆ demande à l'utilisateur de fournir des fonctions d'élévation pour chaque passage entre le dynamique et le statique. Cela permet de supprimer le mécanisme de synchronisation utilisé. Ce travail est toujours en cours.

Concernant la preuve de non-interférence, nous avons découvert au fur et à mesure de son implémentation de nombreuses limitations. Si l'automatisation semble possible, sur un plan technique elle n'est pas praticable. Par exemple, dans certains cas le raffinement est perdu et dans d'autres cas la précondition la plus faible l'est également.

En effet, la fonction d'effacement en Meta⋆ n'implémente pas de spécification puisque Meta⋆ ne le permet pas. De plus, écrire une spécification pour une fonction effacée peut être non trivial puisqu'il faut prendre en compte les valeurs de retour en fonction de l'effacement en cours. Cela pose un problème lorsque l'on souhaite vérifier des prédicats sur un contexte.

Pour contourner cette limitation, nous définissons les effets de manière totale, ce qui permet d'"exécuter" l'effet en utilisant la normalisation de F⋆ (Marty et al. 2020). Ce dernier point est très gourmand en ressources et de temps en temps il échoue, car la normalisation n'est pas suffisamment appliquée. En fait, dans la section précédente, nous ne parlons pas de l'effacement du contexte. Bien que l'effacement du contexte soit primordial pour vérifier la non-interférence pour les effets de bord, les problèmes rencontrés en rendent difficile l'utilisation dans des cas simples. Cependant, dans notre article (Marty et al. 2020) nous discutons de cette limitation.

Nous nous sommes rendu compte qu'en l'état, F⋆ n'est pas un outil approprié pour automatiser une preuve comme la non-interférence comme nous le faisons. Cependant, nous soulignons qu'il est entièrement possible d'implémenter cette automatisation de la preuve par simulation au niveau d'un compilateur pour la rejouer sur des programmes à la compilation. Il serait très intéressant de vérifier cette propriété sur des programmes annotés et écrits en langage C.

Enfin, notre bibliothèque LIO⋆ – une fois extraite en C – est partiellement utilisable par un programme en C. En effet, le papier (Stefan, Russo, Mitchell et al. 2017) présente un opérateur `toLabeled` pour résoudre le problème de contamination des labels. Si cette contribution est très importante sur le plan théorique, elle l'est beaucoup moins sur le plan technique. Cette solution suppose que l'opération protégée par `toLabeled` n'a aucun effet de bord sur le CFI. Cela est vérifiable en Haskell puisque les effets sont construits au travers des monades. En F⋆, nous pouvons également assurer l'absence d'effets de bord par modélisation. Toutefois, cela est probablement impossible au niveau système ou au niveau du langage C. Une piste que nous n'avons pas abordée dans notre travail serait de formellement explorer l'impossibilité – ou la possibilité – d'une solution en l'absence de cet invariant. Une autre piste que nous n'avons pas abordée concerne l'envirronement d'exécution du programme. En effet, un paradigme différent dans le système d'exploitation pourrait-il quantifier l'effet de bord d'un programme ?

Conclusion

Nous présentons dans cette conclusion un bref résumé du plan du document afin de rappeler le cheminement de ce manuscrit. Ensuite, nous listons les différentes contributions qui nous permirent de répondre aux questions de recherche. Toutefois, nous présentons de nouvelles pistes de recherche que nous avons identifiées au cours de notre travail. Enfin, nous terminons par une synthèse très globale de ce travail.

Notre travail présente les deux langages que nous avons utilisés dans la partie I. Cela nous a permis de donner d'abord un aperçu de la réalité de l'informatique embarquée au travers du langage C dans le chapitre 1. Puis, nous avons montré ce que F⋆ apporte en termes de modélisation à travers les chapitres 2 et 3. Nous connectons ces deux mondes grâce à l'extraction en langage C permis par F⋆ via le compilateur KreMLin dont nous discutons les avantages et les inconvénients dans le chapitre 4. La possibilité de raisonner sur le code écrit en F⋆, grâce au calcul de la précondition la plus faible, nous permet de réduire l'effort de preuve, et l'extraction en C nous permet d'avoir une application directe de notre programme vérifié.

Nous exploitons ces deux avantages du langage F⋆ dans la partie II. Dans le chapitre 5, nous avons d'abord modélisé un microcontrôleur – l'Arduino – permettant d'exprimer en F⋆ un programme, puis de l'extraire et de l'exécuter directement sur la machine. Cela nous a permis de développer une approche "*écrire, vérifier et exécuter*" depuis un environnement prouvé vers une architecture système. Ensuite, nous avons appliqué dans le chapitre 6 cette approche à deux cas d'étude : un moteur pas à pas qui illustre la vérification de systèmes *cyberphysiques* en section 6.1, et un gestionnaire de mémoire virtuelle qui démontre la vérification de composant système appliquée à l'embarqué en section 6.2.

Nous abordons dans la partie III le CFI. Nous avons proposé une implémentation de la bibliothèque LIO en F⋆ qui peut s'extraire en langage C dans le chapitre 7. Cette version modélise l'effet de programme et nous l'illustrons dans le chapitre 8 par un gestionnaire de mémoire partagée où le CFI sert à contrôler la propagation de l'information. Nous proposons également une approche peu utilisée : la labellisation statique à gros grain. Complètement contre-intuitif, ce type de labellisation permet d'alléger un programme vérifié initialement dynamiquement en le transformant en programme vérifié statiquement. Décrit dans le chapitre 9, ce LIO hybride permet d'écrire un programme, puis de choisir le type de vérification voulu (statique ou dynamique) en fonction du rapport de coût entre performance et effort de preuve. Cela montre qu'il est possible de combiner techniquement la vérification statique et la vérification dynamique dans un même programme embarqué. Nous montrons aussi qu'il est possible d'utiliser LIO dans un contexte embarqué. En effet, nous pouvons modéliser l'impact d'un bloc de code et donc utiliser la solution de LIO sur le problème de contamination des labels.

Dans le chapitre 10 de cette même partie, nous étendons la preuve de la non-interférence d'un programme en combinant la métaprogrammation et le calcul de la précondition la plus faible. De cette manière, nous tentons de prouver une instance d'un programme pour toutes entrées et tous paramètres possibles. Bien qu'elle soit limitée par l'implémentation du calcul de la précondition la plus faible en F⋆, la métaprogrammation met en avant une nouvelle approche qui donne l'espoir d'une preuve automatisée de la non-interférence.

Contributions

Ce travail de recherche a donné lieu à plusieurs contributions que nous décrivons dans ce manuscrit. Cette section liste succinctement les contributions que nous avons produites dans les deux axes de recherche choisis : l'informatique embarquée en partie II et le CFI en partie III. Nous avons réalisé des contributions dans chacun des deux axes à savoir :

- À propos de l'informatique embarquée en partie III :
  - Une modélisation de l'Arduino présenté en chapitre 5 (§5.1),
  - Une modélisation et une preuve de CPS décrites en chapitre 6 (§6.1),
  - Une reproduction d'un MMU de l'état de l'art exposer en chapitre 6 (§6.2),
- Au sujet du CFI en partie III :
  - Une implémentation LIO une bibliothèque de CFI anciennement implémentée en Haskell,
  - Une combinaison d'une approche CFI statique et dynamique formellement vérifiée,
  - Une automatisation de preuve limitée de la propriété de non-interférence.

Ces contributions nous permettent de répondre aux questions de recherche et d'en résumer notre point de vue sur leur argumentation, ce que nous faisons à la section suivante.

Réponses aux questions de recherche

Dans cette section, nous reprenons les questions de recherche posées en introduction et nous apportons une réponse synthétique. L'objectif est de donner une réponse simple, mais représentative de nos résultats qui répond à chacune de ces questions de recherche.

1. **Embarqué** : *pouvons-nous modéliser un état interne de la carte Arduino et rejeter tout programme qui ne respecterait pas les contraintes de cet état ?*

   Cette simple question peut paraître évidente, mais modéliser une telle architecture en F⋆ lève plusieurs problèmes. D'abord, la modélisation des entrées/sorties est dépendante de leur utilisation physique. Puis, la modélisation de l'architecture est limitée puisqu'elle ne prend pas en compte le fait d'interagir avec l'environnement. Enfin, la spécification donnée par le constructeur ne garantit pas l'implémentation du microcontrôleur. Ces limites indiquent quels points peuvent réduire l'assurance qu'apporte la vérification formelle,

mais cela nous donne l'espoir que la vérification des invariants de plus en plus complexes en tenant compte du matériel est techniquement viable.

2. ***Cyberphysique*** : *nous est-il possible d'anticiper l'évolution physique d'un système en fonction du logiciel et de rejeter tout programme qui contournerait la spécification ?*

   En tenant compte de la question précédente, nous avons voulu montrer comment F⋆ peut être utilisé pour anticiper les valeurs possibles d'un programme en tenant compte de son impact sur l'environnement. Cette question est très vaste et nous avons décidé de prendre un exemple simple : le décrochage d'un moteur pas à pas. Le langage F⋆ ne permet pas de modéliser des problèmes physiques continus. Cette réponse confirme que la vérification de l'informatique embarquée est difficile puisque dans notre exemple, nous avons éludé de nombreuses variables comme la charge du moteur. Cela nous confirme que les CPS sont complexes à vérifier, mais possible pour des problèmes spécifiques.

3. **Praticabilité** : *quelle est la difficulté d'implémenter un outil système de l'état de l'art ?*

   Une piste que nous n'avons finalement pas abordée, une implémentation de l'état de l'art a été modélisée en F⋆. L'objectif initial était de comparer l'isolation mémoire avec un algorithme classique et avec un algorithme basé sur un CFI. Toutefois, nous sommes convaincus de la faisabilité de la vérification de fonctionnalités système comme un MMU.

4. **CFI** : *comment garantir un CFI correct dans un environnement système tel que proposé par la programmation en langage C ?*

   Cette question a été compliquée à aborder. Dans la programmation système comme dans un langage de bas niveau (C, Assembleur…), il n'existe aucun moyen de contrôler l'effet de bord contrairement à des langages de haut niveau (Haskell, Java…). Deux pistes s'offraient à nous : le contrôle dynamique par virtualisation comme proposé par Java ou la vérification statique comme proposée par Rust. F⋆ s'inscrit dans cette dernière piste. Toutefois, le code une fois extrait en C peut être en contact avec un code non vérifié et cela peut détruire toute assurance apportée par la vérification. Dans ce cas, une isolation système (par exemple un MMU matériel) est nécessaire, ce que nous contournons en informatique embarquée par l'hypothèse que la vérification couvre tout le système. Sans cette hypothèse, ce travail n'apporte aucune garantie par rapport à une implémentation non vérifiée.

5. **Mixité d'approches** : *comment avoir un positionnement de la vérification des invariants qui peut être soit dynamique soit statique ?*

   Dans nos travaux, nous avons constaté que la vérification statique possède de meilleures performances en dépit d'une prouvabilité plus difficile. Nous avons réussi à combiner les deux formes de vérification de manière à avoir le choix de la forme la plus accessible. Cependant, nous avons introduit une approximation lors du passage d'un code vérifié statiquement vers un code vérifié dynamiquement. Cette approximation est manuellement formulée par le développeur, mais reste vérifiée automatiquement.

6. **Automatisation** : *comment automatiser une preuve de la non-interférence d'un système de CFI grâce à la métaprogrammation ?*

   Travailler avec le langage F⋆ nous a amenés à utiliser le calcul de la précondition la plus faible pour inférer des propriétés dans un programme. Puisque le langage F⋆ peut vérifier automatiquement des théorèmes générés via Meta⋆, nous avons tenté de le faire en générant une fonction identique, mais non interférente par construction. Si sur des programmes simples cela fonctionne, nous nous sommes heurtés à des limitations techniques du langage. En revanche, nous restons positifs sur la génération d'obligations de preuve et la preuve automatique.

Nouvelles questions de recherche

Les questions de recherche que nous décrivons dans la section précédente ont donné lieu à de nouvelles questions. Bien qu'elles sont nombreuses, ces questions ne sont qu'un sous-ensemble des questions que nous nous sommes posées pendant notre travail.

1. **CPS** : *Automatisation des preuves CPS.*
   — Dans nos travaux, nous présentons une preuve de CPS. Cependant, F⋆ n'est pas conçu pour cette tâche, car la modélisation de CPS se fait sur une arithmétique continue comme la logique différentielle dynamique. Nous pensons qu'il est possible de construire un langage qui permet à la fois de faire des preuves sur un programme en utilisant des données discrètes et des données continues. Un tel langage permettrait d'exprimer plus efficacement des propriétés physiques et donc simplifierait la preuve de CPS.
   — Nous avons noté la complexité apportée par les interactions physiques dans la preuve de programmes. Cependant, est-il possible de quantifier cette complexité ? Répondre à cette question donnera une idée de la faisabilité de la preuve des CPS.
2. **MMU** : *Comparatif des performances et de l'expressivité.*
   — Nous avons développé deux MMU différents : l'un basé sur une conception classique présente dans l'état de l'art et l'autre basé sur un CFI pour isoler les pages mémoires des processus dans un système. Outre le fait que ce dernier MMU permet le partage de pages mémoire, les deux algorithmes d'isolation sont très différents. Quels sont leurs avantages et leurs inconvénients respectifs ? En effet, l'approche labellisée apporte-t-elle un gain ?
   — Dans notre modélisation d'un MMU de l'état de l'art, nous avons montré comment des mécaniques système sont implémentées. Si moralement la vérification n'a pas de coût à l'exécution puisqu'elle est effacée à la compilation, nous n'avons pas comparé les performances entre les deux versions. De plus, les propriétés vérifiées dans notre

implémentation sont faibles : nous vérifions juste que l'algorithme correspond à la spécification donnée, mais nous ne vérifions pas que l'isolation est correcte.

3. **LIO** : *Application à l'informatique système.*
   — Notre travail étend l'état de l'art de LIO pour l'adapter à une problématique système. Se pourrait-il que LIO puisse être utilisée pour prévenir la fuite d'information dans un système d'exploitation ? Notre travail couvrant les systèmes de type *libOS*, quel paradigme système serait le plus approprié ?
   — L'adaptation a montré la complexité à prendre en compte l'hypothèse de l'absence d'effet de bord. En effet, c'est une question à laquelle nous répondons en contournant le problème. Dans l'industrie, ce problème est résolu soit par la virtualisation, soit par une implémentation matérielle du CFI. Un paradigme de système d'exploitation pourrait-il permettre de résoudre ce problème ?
   — Dans notre implémentation d'un LIO hybride, nous avons dû implémenter un mécanisme pour maintenir synchroniser le label effacé et le label dynamique. Cette solution n'est pas élégante et avec les avancées du langage F⋆, il serait agréable de voir une implémentation plus élégante qui pourrait en plus mettre à jour le label dynamique automatiquement. Cette avancée permettrait d'être plus proche de HLIO.

4. **Automatisation** : *Extension de la preuve de non-interférence automatique.*
   — Il nous apparaît que cette automatisation peut être appliquée à d'autres langages comme le Java. Ainsi, nous pourrions imaginer une annotation indiquant si une information est dite sensible et automatiquement vérifier si un bloc de code est non interférent. Dans une telle situation, quels problèmes cela pourrait-il lever ? Est-ce qu'un calcul de la précondition la plus faible est adaptable à des langages industriels (C, JAVA, Rust) ?
   — Une telle automatisation de preuve pourrait également être appliquée à de l'analyse statique de binaire pour déterminer si un programme est non interférent. Toutefois, comme une analyse à cette échelle peut être plus complexe, est-ce qu'une telle solution est praticable en termes de performance ?
   — Dans nos travaux, nous avons utilisé le calcul de la précondition la plus faible. C'est un fonctionnement central du langage F⋆. Si dans de futurs travaux, nous développons une autre solution, est-ce que dans la propagation des labels, le calcul de la postcondition la plus forte est plus efficace ?

Synthèse

Le monde de l'informatique système et embarqué peut-il bénéficier d'un contrôle de flux vérifié en fonction du matériel et de l'environnement ? Et face à la verbosité des programmes système, pouvons-nous utiliser le modèle "*écrire, vérifier et exécuter*" ?

La réponse à ces deux questions est en partie avancée dans notre travail. Nous avons poussé la modélisation jusqu'à l'état des entrées/sorties. Bien sûr, une modélisation de l'architecture du microcontrôleur que nous utilisons serait plus précise, mais nous ne pourrions pas adapter notre modélisation sur d'autres microcontrôleurs et de plus l'Atmel328p n'est pas open source. Nous avons également montré comment modéliser l'état de l'environnement tel qu'il est impacté par notre programme. Cela révèle bien qu'un outil comme F⋆ peut être utilisé pour des systèmes *cyberphysiques*. De plus, notre travail souligne la faisabilité – toujours plus réaliste – d'une vérification automatisée. Ce travail est convaincant sur l'efficacité du typage dans la vérification de la correction des programmes. Cependant, ce qui permet d'assurer la correction d'un programme est d'abord la bonne compréhension de son comportement, et de fait, toutes les méthodes qui augmentent les connaissances sur le programme permettent d'augmenter les assurances : "*assurer c'est comprendre*".

Nous sommes partis d'une question concrète : comment utiliser des solutions classiques de l'industrie tout en ayant des assurances de propriétés formelles. De l'utilisation réaliste du langage C, nous avons utilisé F⋆ pour abstraire les programmes et leurs contraintes. Cette modélisation nous a permis de valider des invariants pour toutes entrées/sorties et donc de prouver la correction de ces propriétés. L'automatisation de la preuve de programme a été un point décisif, puisque nous pensons que les programmes systèmes ont des preuves verbeuses, ce qui est confirmé par l'utilisation de L⋆. Finalement, nous avons utilisé la chaîne de compilation KreMLin pour appliquer concrètement ces résultats. Un tel retour à la réalité a parfois mis en lumière la complexité parfois sous-estimée de certaines hypothèses comme la garantie d'un contexte consistant dans un langage comme le C.

Finalement, le calcul de la précondition la plus faible avec un solveur SMT permet de simplifier grandement la vérification et cela nous permet de nous approcher d'un modèle "*écrire, vérifier et exécuter*". De plus, la modélisation à la fois d'un microcontrôleur, de l'environnement et du CFI permet de garantir la vérification de propriétés telles que la correction des interactions avec le monde physique ou la non-interférence au niveau des systèmes embarqués et des systèmes *cyberphysiques*.

Références

A High-Assurance Cryptographic Library (2020). URL : https://github.com/project-everest/hacl-star (visité le 02/09/2020).

Ahman, Danel et al. (2017). « Dijkstra monads for free ». In : *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, p. 515-529.

Algehed, Maximilian, Jean-Philippe Bernardy et Catalin Hritcu (2020). « Dynamic IFC Theorems for Free ! » In : *arXiv preprint arXiv :2005.04722*.

Arduino Documentation - analog read (2020). URL : https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/ (visité le 23/03/2020).

Ariola, Zena M et Amr Sabry (1998). « Correctness of monadic state : An imperative call-by-need calculus ». In : *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 62-74.

Atmel-7810-Automotive-Microcontrollers-ATmega328P Datasheet (2020). URL : http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (visité le 27/07/2020).

Austin, Thomas H et Cormac Flanagan (2009). « Efficient purely-dynamic information flow analysis ». In : *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, p. 113-124.

Behm, Patrick et al. (1999). « METEOR : A successful application of B in a large project ». In : *International Symposium on Formal Methods*. Springer, p. 369-387.

Bell, D Elliott et Leonard J LaPadula (1973). *Secure computer systems : Mathematical foundations*. Rapp. tech. MITRE CORP BEDFORD MA.

Besson, Frédéric, Nataliia Bielova et Thomas Jensen (2013). « Hybrid information flow monitoring against web tracking ». In : *2013 IEEE 26th Computer Security Foundations Symposium*. IEEE, p. 240-254.

— (2016). « Hybrid monitoring of attacker knowledge ». In : *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, p. 225-238.

Bhargavan, Karthikeyan et al. (2013). « Implementing TLS with verified cryptographic security ». In : *2013 IEEE Symposium on Security and Privacy*. IEEE, p. 445-459.

Buiras, Pablo, Dimitrios Vytiniotis et Alejandro Russo (2015). « HLIO : Mixing static and dynamic typing for information-flow control in Haskell ». In : *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, p. 289-301.

C (langage) (2020). URL : https://fr.wikipedia.org/wiki/C_(langage) (visité le 26/08/2020).

Camposano, Raul et Jörg Wilberg (1996). « Embedded system design ». In : *Design Automation for Embedded Systems* 1.1-2, p. 5-50.

Chip Hall of Fame : Atmel ATmega8 (2017). URL : https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-atmel-atmega8 (visité le 13/03/2020).

Choudhuri, Siddharth et Tony Givargis (2005). « Software virtual memory management for MMU-less embedded systems ». In : *Center for Embedded Computer Systems*, p. 12.

Cohen, Ellis (1977). « Information transmission in computational systems ». In : *Proceedings of the sixth ACM symposium on Operating systems principles*, p. 133-139.

Colnaric, Matjaz (1999). « State of the art review paper : Advances in embedded hard real-time systems design ». In : *ISIE'99. Proceedings of the IEEE International Symposium on Industrial Electronics (Cat. No. 99TH8465)*. T. 1. IEEE, p. 37-42.

Cummings, David M (2016). « Embedded software under the courtroom microscope : A case study of the Toyota unintended acceleration trial ». In : *IEEE Technology and Society Magazine* 35.4, p. 76-84.

Denning, Dorothy E (1976). « A lattice model of secure information flow ». In : *Communications of the ACM* 19.5, p. 236-243.

Dijkstra, Edsger W. (août 1975). « Guarded Commands, Nondeterminacy and Formal Derivation of Programs ». In : *Commun. ACM* 18.8, p. 453-457. ISSN : 0001-0782. DOI : 10.1145/360933.360975. URL : https://doi.org/10.1145/360933.360975.

Enck, William et al. (2014). « TaintDroid : an information-flow tracking system for realtime privacy monitoring on smartphones ». In : *ACM Transactions on Computer Systems (TOCS)* 32.2, p. 1-29.

Espiau, Bernard, Konstantinos Kapellos et Muriel Jourdan (1996). « Formal verification in robotics : Why and how ? » In : *Robotics Research*. Springer, p. 225-236.

everquic-crypto (2020). URL : https://github.com/project-everest/everquic-crypto (visité le 02/09/2020).

Ferraiuolo, Andrew et al. (2018). « HyperFlow : A processor architecture for nonmalleable, timing-safe information flow security ». In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, p. 1583-1600.

Fromherz, Aymeric et al. (2019). « A Verified, Efficient Embedding of a Verifiable Assembly Language ». In : *Principles of Programming Languages (POPL 2019)*. ACM. URL : https://www.microsoft.com/en-us/research/publication/a-verified-efficient-embedding-of-a-verifiable-assembly-language/.

Goguen, Joseph A et José Meseguer (1982). « Security policies and security models ». In : *1982 IEEE Symposium on Security and Privacy*. IEEE, p. 11-11.

Guaspari, David, Carla Marceau et Wolfgang Polak (1990). « Formal verification of Ada programs ». In : *IEEE Transactions on Software Engineering* 16.9, p. 1058-1075.

Harrison, Michael A, Walter L Ruzzo et Jeffrey D Ullman (1976). « Protection in operating systems ». In : *Communications of the ACM* 19.8, p. 461-471.

Hathhorn, Chris, Chucky Ellison et Grigore Roşu (2015). « Defining the undefinedness of C ». In : *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 336-345.

Hauer, Barbara (2015). « Data and information leakage prevention within the scope of information security ». In : *IEEE Access* 3, p. 2554-2565.

Hirsch, Andrew K et Ethan Cecchetti (2020). « Giving Semantics to Program-Counter Labels via Secure Effects ». In : *arXiv preprint arXiv :2010.13191*.

Hoare, Charles Antony Richard (1969). « An axiomatic basis for computer programming ». In : *Communications of the ACM* 12.10, p. 576-580.

Jiménez, Manuel, Rogelio Palomera et Isidoro Couvertier (2013). *Introduction to embedded systems*. Springer.

Johnston, Phillip et Rozi Harris (2019). « The Boeing 737 MAX saga : lessons for software organizations ». In : *Software Quality Professional* 21.3, p. 4-12.

Kocher, Paul et al. (2019). « Spectre attacks : Exploiting speculative execution ». In : *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, p. 1-19.

Koopman, Phil (2020). *A case study of Toyota unintended acceleration and software safety*. url : http://webhost.laas.fr/TSF/IFIPWG/Workshops&Meetings/69/Workshop-regularPapers/Koopman14_toyota_ua_slides.pdf (visité le 27/07/2020).

Kosmatov, Nikolai et Julien Signoles (2016). « Frama-C, A collaborative framework for C code verification : Tutorial synopsis ». In : *International Conference on Runtime Verification*. Springer, p. 92-115.

Krohn, Maxwell et al. (2007). « Information flow control for standard OS abstractions ». In : *ACM SIGOPS Operating Systems Review* 41.6, p. 321-334.

Kuzminykh, Ievgeniia et Maryna Yevdokymenko (2019). « Analysis of Security of Rootkit Detection Methods ». In : *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*. IEEE, p. 196-199.

Leroy, Xavier (2009). « Formal verification of a realistic compiler ». In : *Communications of the ACM* 52.7, p. 107-115.

Li, Peng et Steve Zdancewic (2010). « Arrows for secure information flow ». In : *Theoretical computer science* 411.19, p. 1974-1994.

Lipp, Moritz et al. (2018). « Meltdown ». In : *arXiv preprint arXiv :1801.01207*.

List of Arduino boards and compatible systems (2020). url : https://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems (visité le 13/03/2020).

Loghi, Mirko et al. (2005). « Dynamic and formal verification of embedded systems : A comparative survey ». In : *International Journal of Parallel Programming* 33.6, p. 585-611.

Lomont, Chris (2003). « Fast inverse square root ». In : *Tech-315 nical Report* 32.

Losing Face : Two More Cases of Third-Party Facebook App Data Exposure (2020). URL : https://www.upguard.com/breaches/facebook-user-data-leak/ (visité le 28/09/2020).

MAILLARD, Kenji et al. (2019). « Dijkstra Monads for All ». In : *Proceedings of the ACM on Programming Languages*.

Mandatory access control (2020). URL : https://en.wikipedia.org/wiki/Mandatory_access_control (visité le 24/11/2020).

MAO, Ziqing et al. (2009). « Trojan horse resistant discretionary access control ». In : *Proceedings of the 14th ACM symposium on Access control models and technologies*, p. 237-246.

MARTÍNEZ, Guido et al. (2019). « Meta-F* : Proof Automation with SMT, Tactics, and Metaprograms ». In : *European Symposium on Programming*. Springer, Cham, p. 30-59.

MARTY, Jean-Joseph et al. (2020). « LIO* : Low Level Information Flow Control in F ». In : *arXiv preprint arXiv :2004.12885*.

MARWEDEL, Peter (2006). *Embedded system design*. T. 1. Springer.

miTLS : A Verified Reference Implementation of TLS (2020). URL : https://www.mitls.org/ (visité le 02/09/2020).

MYERS, Andrew C (1999). « JFlow : Practical mostly-static information flow control ». In : *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 228-241.

PARKER, James, Niki VAZOU et Michael HICKS (2019). « LWeb : Information flow security for multi-tier web applications ». In : *Proceedings of the ACM on Programming Languages* 3.POPL, p. 1-30.

Playing with Arduino (2020). URL : https://garretlab.web.fc2.com/en/arduino/inside/ (visité le 19/03/2020).

Port Registers (2020). URL : https://www.arduino.cc/en/Reference/PortManipulation (visité le 13/03/2020).

POTTIER, François et Sylvain CONCHON (2000). « Information flow inference for free ». In : *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, p. 46-57.

Predicate transformer semantics (2020). URL : https://en.wikipedia.org/wiki/Predicate_transformer_semantics (visité le 02/09/2020).

Proof-oriented Programming in F - Universes* (2022). URL : https://www.fstar-lang.org/tutorial/book/part2/part2_universes.html.

PROTZENKO, Jonathan (2018). *The KreMLin user manual and documentation*. URL : https://fstarlang.github.io/lowstar/ (visité le 01/01/2020).

PROTZENKO, Jonathan, Benjamin BEURDOUCHE et al. (2019). « Formally verified cryptographic web applications in WebAssembly ». In : *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, p. 1256-1274.

PROTZENKO, Jonathan, Jean-Karim ZINZINDOHOUÉ et al. (2017). « Verified low-level programming embedded in F* ». In : *Proceedings of the ACM on Programming Languages* 1.ICFP, p. 17.

QUIC (2020). URL : https://en.wikipedia.org/wiki/QUIC (visité le 02/09/2020).

Racine carrée inverse rapide (2020). URL : https://fr.wikipedia.org/wiki/Racine_carr%C3%A9e_inverse_rapide (visité le 31/08/2020).

RICHARDS, Martin (1969). « BCPL : A tool for compiler writing and system programming ». In : *Proceedings of the May 14-16, 1969, spring joint computer conference*, p. 557-566.

ROBINSON, Ken (2011). *Ariane 5 : flight 501 failure a case study*.

ROY, Indrajit et al. (2009). « Laminar : Practical fine-grained decentralized information flow control ». In : *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 63-74.

RUSSO, Alejandro (2015). « Functional pearl : two can keep a secret, if one of them uses Haskell ». In : *ACM SIGPLAN Notices* 50.9, p. 280-288.

RUSSO, Alejandro, Koen CLAESSEN et John HUGHES (2008). « A library for light-weight information-flow security in Haskell ». In : *ACM Sigplan Notices* 44.2, p. 13-24.

SABELFELD, Andrei et Andrew C MYERS (2003). « Language-based information-flow security ». In : *IEEE Journal on selected areas in communications* 21.1, p. 5-19.

SANDHU, Ravi et Qamar MUNAWER (1998). « How to do discretionary access control using roles ». In : *Proceedings of the third ACM workshop on Role-based access control*, p. 47-54.

SINGH, Neeraj Kumar, Andy WELLINGS et Ana CAVALCANTI (2012). « The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada ». In : *Proceedings of the 10th international workshop on Java technologies for real-time and embedded systems*, p. 62-71.

SINGH, Simon (2000). *The code book : the science of secrecy from ancient Egypt to quantum cryptography*. Anchor.

SOURI, Alireza et Monire NOROUZI (2019). « A state-of-the-art survey on formal verification of the internet of things applications ». In : *Journal of Service Science Research* 11.1, p. 47-67.

STEFAN, Deian, Alejandro RUSSO, David MAZIÈRES et al. (2011). « Disjunction category labels ». In : *Nordic conference on secure IT systems*. Springer, p. 223-239.

STEFAN, Deian, Alejandro RUSSO, John C MITCHELL et al. (2011). « Flexible dynamic information flow control in Haskell ». In : *Proceedings of the 4th ACM symposium on Haskell*, p. 95-106.

— (2017). « Flexible Dynamic Information Flow Control in the Presence of Exceptions ». In :

Stepper Motor Guide (2020). URL : http://www.motioncontrolguide.com/learn/guides/stepper-motor-guide/ (visité le 24/07/2020).

STEWART, Gordon et al. (2015). « Compositional compcert ». In : *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 275-287.

Swamy, Nikhil, Juan Chen et Ben Livshits (2013). « Verifying Higher-order Programs with the Dijkstra Monad ». In : *ACM Programming Language Design and Implementation (PLDI) 2013*. ACM. url : https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/.

Swamy, Nikhil, Cătălin Hriţcu et al. (2016). « Dependent types and multi-monadic effects in F* ». In : *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 256-270.

Talpin, Jean-Pierre et al. (2019). « Towards verified programming of embedded devices ». In : *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, p. 1445-1450.

Team, The F* (2020). *Verified programming in F**. url : https://www.fstar-lang.org/tutorial/ (visité le 17/08/2020).

The Development of the C Language (2020). url : https://www.bell-labs.com/usr/dmr/www/chist.html (visité le 26/08/2020).

Trusted Computer System Evaluation Criteria (2020). url : https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/dod85.pdf (visité le 18/11/2020).

Trusted Computer System Evaluation Criteria (2020). url : https://en.wikipedia.org/wiki/Trusted_Computer_System_Evaluation_Criteria (visité le 24/11/2020).

Vassena, Marco et al. (2019). « From fine-to coarse-grained dynamic information flow control and back ». In : *Proceedings of the ACM on Programming Languages* 3.POPL, p. 1-31.

Vazou, Niki et al. (2014). « Refinement types for Haskell ». In : *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, p. 269-282.

Volpano, Dennis, Cynthia Irvine et Geoffrey Smith (1996). « A sound type system for secure flow analysis ». In : *Journal of computer security* 4.2-3, p. 167-187.

Wang, Jichuan et al. (2007). « Developing an insulin pump system using the SOFL method ». In : *14th Asia-Pacific Software Engineering Conference (APSEC'07)*. IEEE, p. 334-341.

Wang, Peng et al. (p. d.). « Extracting from F* to C : a progress report ». In : ().

What is an RTOS ? (2020). url : https://www.highintegritysystems.com/rtos/what-is-an-rtos/ (visité le 21/07/2020).

What is stepper motor resonance (2020). url : http://www.motioncontrolguide.com/learn/faqs/motors/stepper-motors/what-is-stepper-motor-resonance/ (visité le 24/07/2020).

Witt, Stephen (2020). *Apollo 11 mission out of control*. url : https://www.wired.com/story/apollo-11-mission-out-of-control/ (visité le 05/05/2020).

Xiao, Yuan et al. (2016). « One bit flips, one cloud flops : Cross-vm row hammer attacks and privilege escalation ». In : *25th {USENIX} Security Symposium ({USENIX} Security 16)*, p. 19-35.

ZHANG, Yi et al. (2011). « Generic safety requirements for developing safe insulin pump software ». In : *Journal of diabetes science and technology* 5.6, p. 1403-1419.

ZINZINDOHOUÉ, Jean-Karim et al. (2017). « HACL* : A verified modern cryptographic library ». In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, p. 1789-1806.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Contrôle vérifié de flux d'information appliqué aux systèmes cyberphysiques

Mot clés : Contrôle de flux d'information, Vérification formelle, Informatique embarquée

Résumé : L'informatique est devenue omniprésente dans la vie de tous les jours au travers d'outils de plus en plus riches technologiquement et de plus en plus connectés à internet. Lorsque ces objets arrêtent de fonctionner suite à un concours de circonstances, cela peut avoir des conséquences dramatiques sur la sûreté des personnes qui en dépendent. Leur fonctionnement peut être aussi altéré par une volonté malveillante de nuire et qui lors d'une attaque créera la situation propice à un dysfonctionnement ciblé.
Dans ce travail, nous explorons l'utilisation des méthodes formelles dans un environnement embarqué et cyberphysique. Dans un premier temps, nous utilisons F⋆ pour modéliser la sûreté des programmes sur une Arduino. Puis nous implémentons LIO (Library Input Output) en F⋆ dans un contexte système ou embarqué. Nous mettons ici en avant une approche qui permet de choisir entre une vérification statique et une vérification dynamique afin de réduire la charge de preuve ou le coût à l'exécution. Nous proposons également une automatisation de la preuve de non-interférence en utilisant la métaprogrammation. Cette preuve porte sur l'interaction entre le programme et la bibliothèque plutôt que sur la bibliothèque seulement.

Title: Verified information flow control applied to cyber-physical systems

Keywords: Information flow control, formal verification, embedded systems

Abstract: Computing has become omnipresent in daily life through tools that are technologically enriched and increasingly connected to the Internet. When these objects stop working due to a combination of circumstances, it can have dramatic consequences on the safety of people who depend on them. Their functioning can also be altered by a malicious will to harm and which, during an attack, will create a situation conducive to a targeted malfunction.
In this work, we explore the use of formal methods in an embedded and cyber-physical environment. First, we use F⋆ to model the safety of programs on an Arduino. In a second step, we implement LIO (Library Input Output) in F⋆ in a system or embedded context. We put forward an approach that allows to choose between a static verification and a dynamic verification in order to reduce the proof load or the cost at runtime. We also propose a mechanization of the non-interference proof using metaprogramming. This proof focuses on the interaction between the program and the library rather than on the library only.