Programmation vérifiée à l'intersection des types dépendants et de l'analyse statique

Lucas Franceschino

THÈSE DE DOCTORAT DE

L'ÉCOLE NORMALE
SUPÉRIEURE RENNES

ÉCOLE DOCTORALE Nº 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par
Lucas FRANCESCHINO

Verified Programming at the Intersection of Dependent Types and Static Analysis

Thèse présentée et soutenue à Inria Rennes, le 10 décembre 2021
Unité de recherche : INRIA

Rapporteurs avant soutenance :

Nikhil SWAMY	Senior principal researcher à Microsoft Research
Stephan MERZ	Directeur de recherche à Inria Nancy

Composition du Jury :

Président :	Sandrine BLAZY	Professeur à Université de Rennes 1
Examinateurs :	Nikhil SWAMY	Senior principal researcher à Microsoft Research
	Stephan MERZ	Directeur de recherche à Inria Nancy
	Sandrine BLAZY	Professeur à Université de Rennes 1
	Karthikeyan BHARGAVAN	Directeur de recherche à Inria Paris
	Pierre-Yves STRUB	Maître de conférence à l'École Polytechnique
	Niki VAZOU	Assistant Research Professor à IMDEA Software Institute
Dir. de thèse :	Jean-Pierre TALPIN	Directeur de recherche à Inria Paris
Co-enc. de thèse :	David PICHARDIE	Chercheur Facebook Research

Contents

Remerciements

"*All things are difficult before they are easy.*"

Cela me parait particulièrement adapté au long voyage tortueux qu'ont été mes trois années de thèse, voyage qui aurait été bien trop assommant sans l'aide d'un bon nombre de personnes, que je tiens à remercier en préambule de ce manuscrit.

Tout d'abord, mes remerciements vont à mon directeur de thèse Jean-Pierre Talpin, qui m'a soutenu tout au long de ma thèse et rassuré dans les moments plus difficiles. Puis, naturellement, je veux aussi remercier David Pichardie, mon co-encadrant de thèse et *formidable consultant en interprétation abstraite*, avec qui j'ai eu grand plaisir à travailler, et qui m'a donné de précieux conseils. Merci pour vos précieuses relectures détaillées, et pour tout le temps que l'on a pu passer à discuter, et notamment à retourner dans tous les sens mes différentes constructions et formalisations de monades hybrides.

I would also like to thank Sandrine Blazy, Karthikeyan Bhargavan, Pierre-Yves Strub, and more particularly Nikhil Swamy and Stephan Merz for their report. It was an honour to have you in my thesis committee.

Thank you also Niki Vazou, not only for being a member of my thesis committee, but also for the numerous discussions and Skype calls we had with Jean-Joseph Marty, while designing LIO. It was great to work with you!* Merci aussi à toi Jean-Joseph, pour toutes ces discussions –parfois trop– enflammées, dans notre bureau ou autour de bières.

Cette thèse a été nourrie et ponctuée de nombreuses interactions avec mes collègues et mes amis de Rennes. Je voudrais vous remercier tous pour ces nombreux cafés bus en pause, toutes ces discussions au détour d'un bureau, tous ces repas réchauffés dans les micro-ondes de la cafétéria, toutes ces promenades de confinement, et évidemment toutes ces bières consommées, notamment à l'incroyable *Artiste Assoiffé*. Merci donc à : Stéphane Kastenbaum (*et à notre bonne appréciation du couvre-feu*), Kilian Fatras (*et à Floriane, merci pour vos cadeaux, toujours de très bon goût*), Jérémy Cohen (*et à notre passion commune pour les lapins*), Axel Marmoret (*et à ton incroyable ponctualité*), Jean-Joseph Marty (*malgré ton attrait obscur pour C*), Liangcong Zhang (*and to our epic trip to Morocco*), Younès Zine (*et à toutes ces soirées que l'on commençait à l'Artiste et que tu finissais à la Batchata*), Antoine Chatalic (*et à toutes ces discussions à propos du* pâte-otto*, sa nomenclature et son existence*), Simon Lunel (*et à ces bières brassées chez toi*), mais aussi à Alexandre Honorat, Mathias Malandain, Diego Dicarlo, Cassio Fraga Dantas, Rémi Cornillet, et j'en oublie probablement.

Merci aussi à tous mes amis d'ailleurs, ainsi qu'à ma famille. Merci à mon père et à nos nombreuses heures de procrastination à discuter

au téléphone, ainsi qu'à ma mère à qui j'ai infligé de très nombreuses relectures de mes différents travaux. Merci à toi Lou, pour ton soutien et pour tous ces moments passés ensemble, moments empreints de ton optimisme indéfectible, sans lequel j'aurais probablement abandonné une bonne demi-douzaine de fois. Merci aussi à mon lapin Puggy (évidemment), ainsi *qu'au Julius* et *au Rien* qui se reconnaîtront.

Résumé des travaux en français

Notre vie quotidienne est de plus en plus imprégnée par de nombreux logiciels interconnectés. On discute avec nos amis, notre famille et nos collègues via des applications de messagerie instantanée, nous planifions nos vacances avec Google Maps, partageons nos derniers repas sur Instragram, nous nous divertissons sur Netflix ou YouTube... Le logiciel ressemble un peu à des briques de Lego qui semblent (en apparence) simples et bon marché à assembler, dans l'optique de produire d'autres logiciels plus grands. De cette apparente simplicité résulte une grande complexité logicielle : de nos jours, le moindre logiciel s'avère être un monstre de dépendances logicielles. Par exemple, vérifier simplement le solde de son compte bancaire au moyen de la belle et agréable interface utilisateur affichée par son application bancaire fait en réalité appel à de nombreux et très complexes logiciels, dont probablement certains vieux programmes COBOL[1]. Parmi toutes les constructions humaines, ce sont probablement les logicielles qui sont les plus complexes.

Note: This part is a summary written in French of the manuscript; the rest is written in English and starts with an introduction in Chapter 1.

[1] En 2017, selon l'agence *Reuters*, 43% des systèmes bancaires utilisaient toujours COBOL.

Le moindre incident se produisant dans l'un des composants d'un logiciel peut être suffisant pour provoquer un *bug*. Un bug est un comportement logiciel incorrect, c'est-à-dire un comportement qui n'était pas prévu. Évidemment, les bugs peuvent produire toutes sortes d'effets désagréables. Un exemple de bug est l'accélération incontrôlée des voitures Toyota [Koo14], où des erreurs basiques de programmation ont coûté la vie à plusieurs dizaines de personnes. Sur une note différente, un bug dans un contact Etherum a conduit à la disparition d'environ 50 millions de dollars. L'Institut National des Normes et de la Technologie des États Unis (NIST) a estimé le coût des bugs logiciels à près de 59,5 milliards de dollars chaque année [Pla02].

Les programmes sont exécutés sur des ordinateurs, qui comprennent un langage particulier : le langage machine qui, comme son nom l'indique, est conçu pour être interprété facilement par nos ordinateurs. Ainsi, le langage machine n'est que peu lisible, gpeu pratique et peu productif. Tout comme des ingrédients de qualité subliment un plat délicieux, de bons langages de programmation sont essentiels pour la qualité logicielle. Il existe de très nombreux langages de programmation : la quête pour *Le Meilleur Langage de Programmation* est loin d'être achevée. Pour accomplir cette quête, un premier pas serait de décider d'une métrique pour juger de la qualité d'un langage de programmation. Puisque nous recherchons la meilleure qualité logicielle, dans cette thèse nous sommes intéressés par les langages de programmation qui aident le programmeur à éliminer *systématiquement* les bugs. Nous nous intéressons aux langages équipés de systèmes de typage fort, et qui mettent en œuvre par exemple les types dépendants ou raffinés.

Typage et précision La notion de type est apparue en premier lieu avec le langage de programmation Fortran[2], qui permettait de distinguer les nombres entiers des flottants par exemple. L'entier 42 est représenté par une séquence de bits différente du flottant 42.0 : une addition entière sur des flottants par exemple, produira un résultat incohérent. Puisqu'ils présupposent des représentations, les types des entiers et des flottants sont qualifiés de *primitifs*. Toutefois, les types peuvent être bien plus expressifs, comme en témoignent certains langages de programmation modernes (par exemple, Haskell [Pey07] ou les langages de la famille ML), avec des types inductifs par exemple. Alors que les types primitifs existent pour aider les compilateurs, les types plus avancés aident le programmeur.

[2]Le terme de "type" lui-même n'a été introduit qu'un peu plus tardivement avec le langage Algol.

En effet, plus un programmeur travaille avec des types précis, plus son compilateur lui interdit d'écrire certains programmes incorrects : les types très précis agissent alors comme des spécifications. Au lieu d'écrire un programme en espérant qu'il corresponde à la spécification imaginée, on écrit un programme avec des types traduisant concrètement la spécification en question. Le compilateur[3] vérifie ensuite que le programme corresponde bien au type donné : tout écart vis-à-vis de la spécification provoque alors une erreur de compilation. Selon les systèmes de typage, les types sont plus ou moins expressifs : seuls certains permettent d'écrire des spécifications arbitrairement riches dans leurs types.

[3]Notons que l'on parle ici de typage *statique*, c'est-à-dire que la procédure de vérification du bon typage d'un programme se déroule avant l'exécution de celui-ci.

Expressivité des types et contraintes. Malheureusement, plus un système de type autorise d'expressivité, plus la procédure de vérification des types devient indécidable. Par exemple, la procédure de vérification des types d'un système vérifiant la terminaison est trivialement indécidable par le problème de l'arrêt [Chu36; Tur37]. Par conséquent, écrire un programme affublé de types forts requiert fatalement une assistance particulière de la part du programmeur : il ou elle doit alors fournir des indices, des annotations ou même des preuves manuelles au système de typage.

Notre travail est ancré dans cette observation : utiliser des types très expressifs demande du travail manuel supplémentaire. L'un des langages de programmation proposant des types dépendants les plus connus, Coq, n'a pas été conçu comme un langage de programmation généraliste, mais davantage comme un assistant de preuve ; c'est-à-dire un environnement dans lequel on peut écrire des théorèmes et des preuves. Au contraire, Idris ou F* sont des langages de programmation généralistes équipés de systèmes de typages forts et expressifs. F* est particulièrement intéressant, en effet il fait appel à un SMT solver pour alléger l'effort de preuve pour le programmeur. Aussi, F* ressemble beaucoup à OCaml et est équipé d'un système d'effets monadiques très souple, et son système de typage permet l'usage de types raffinés et dépendants. C'est un langage qui a récemment brillé avec le Projet Everest [EVEREST] dont a découlé une série de librairies cryptographiques vérifiées performantes : HACL* [Zin+17], ValeCrypt [Bon+17] et EverCrypt [Pro+20] ; ces librairies sont par exemple utilisées et déployées dans le logiciel Mozilla Firefox.

Analyse statique. Le typage fort n'est pas la seule approche à la vérification formelle. La plupart des programmes intéressants sont bien trop complexes pour être testés pour chaque entrée possible : une solution brillante à ce problème est l'interprétation abstraite [CC77]. Au lieu d'essayer d'exécuter un programme dans un monde concret, infini et complexe, dans l'optique de remarquer des propriétés, l'idée est d'interpréter un programme dans un monde abstrait choisi avec soin pour faire émerger des propriétés pratiques, par exemple la terminaison systématique de toute interprétation.

Dans cette thèse, nous nous intéressons aux interactions entre l'analyse statique (et plus particulièrement l'interprétation abstraite) et les systèmes de typage fort. Nous commençons par étudier la manière dont les types raffinés et dépendants mis en oeuvre dans F⋆ peuvent nous aider à implémenter un interpréteur abstrait vérifié. Ensuite, nous prenons le problème à l'envers : partant d'un interpréteur abstrait vérifié, par quel moyen pouvons-nous améliorer l'inférence de type dans un système de type tel que celui de F⋆ ? Nous répondons à cette question par la présentation d'un système de transformation de monade de pré-condition la plus faible. Enfin, nous nous penchons sur la manière dont il est possible de vérifier l'analyse de flux d'informations (*Information Flow*) à l'aide de types forts.

Nous nous proposons de résumer les travaux menés dans cette thèse en suivant l'organisation du manuscrit.

Chapitre 2. Le chapitre 2 est une introduction au langage de programmation F⋆, dans lequel nous formalisons les différents travaux de cette thèse. Cette introduction à F⋆ démarre par une brève section sous la forme d'un tutoriel, passant en revue sa syntaxe et ses concepts fondamentaux. Elle se poursuit ensuite avec une présentation détaillée du système d'effets de F⋆, en expliquant le principe des monades de calcul, de spécification, des monades indexées et des monades de pré-condition la plus faible.

Chapitre 3. Les interpréteurs abstraits sont des outils d'analyse statique permettant d'inférer automatiquement des propriétés sur un programme. L'interprétation abstraite est une théorie d'approximation correcte des programmes : si les algorithmes d'interprétation abstraite infèrent qu'un programme respecte une certaine propriété, on a une garantie mathématique. Malheureusement, la plupart des interpréteurs abstraits, bien qu'ils suivent les algorithmes très solides donnés par l'interprétation abstraite, sont sujets à des bugs d'implémentation, mettant à mal leurs garanties. Ainsi, certains interpréteurs abstraits ont été vérifiés formellement en utilisant l'assistant de preuve Coq. Vérifier de tels programmes demande une expertise avec Coq, et requiert l'écriture de longues, complexes et peu accessibles preuves manuelle en Coq. Écrire un interpréteur abstrait vérifié avec Coq demande beaucoup de temps et d'expertise : par exemple, l'interpréteur abstrait vérifié Verasco a demandé environ 17.000 lignes [Jou+] de preuves manuelles Coq.

Dans le chapitre 3, nous montrons qu'il est possible d'écrire un interpréteur abstrait accessible à un public non expert en assistants de preuves. En effet, le chapitre 3 présente presque tout le code source de l'interpréteur :

95 % des 527 lignes de son code. Pensée et construite avec les fonctionnalités d'automatisation de F⋆ en tête, notre implémentation ne fait appel qu'à très peu de preuves manuelles[4]. Dans le chapitre 3.2, nous définissons le langage impératif jouet IMP, équipé d'une mémoire faisant correspondre des noms de variables vers des entiers machines signés, d'opérateurs binaires, d'assignations, de choix non déterministes, de séquences et de boucles. Nous donnons ensuite une sémantique opérationnelle à IMP dans le chapitre 3.3. Nous définissons la notion de domaine abstrait dans le chapitre 3.4, puis du domaine abstrait numérique des intervalles en 3.5. Le chapitre 3.6 formalise en F⋆ la notion d'opérateurs d'élargissement, tandis que les chapitres 3.7 et 3.8 définissent une mémoire et un domaine de mémoire abstrait. Enfin, le chapitre 3.9 équipe notre langage IMP avec une sémantique abstraite, donnant lieu à notre interpréteur abstrait vérifié.

[4]Notre interpréteur ne contient qu'un peu moins de 40 lignes de lignes de preuves explicites, soit un peu moins de 8 % de la totalité de son implémentation. C'est un ratio environ dix fois plus bas que les interpréteurs vérifiés existants.

Chapitre 4. Fort du développement du chapitre 3, nous étudions dans le chapitre 4 la manière dont l'interprétation abstraite peut, à son tour, être utilisée pour aider et rendre plus automatique la vérification de programmes avec effets de bord en F⋆. En effet, le système d'inférence de type de F⋆, et ce particulièrement en présence d'effets de bord, n'est pas conçu pour inférer des types particulièrement précis : F⋆ n'est pas capable par exemple d'inférer un invariant de boucle. C'est justement là que l'interprétation abstraite joue tout son rôle : elle est capable d'inférer de tels invariants, réduisant ainsi les annotations nécessaires au bon typage d'un programme. Ainsi, le chapitre 4 propose une méthodologie pour injecter un interpréteur abstrait dans une monade de pré-condition la plus faible.

Après un aperçu général du but poursuivi par notre travail avec le chapitre 4.1, le chapitre 4.2 décrit précisément la forme d'interpréteur abstrait et de monades de pré-condition la plus faible avec lesquelles nous travaillons, ainsi qu'une notion de compatibilité pour un tel couple. Ensuite, le chapitre 4.3 définit un tel couple d'un interpréteur abstrait et d'une monade pour le langage impératif $\text{IMP}^{\times}$, proche du langage IMP défini dans le chapitre 3. Le chapitre 4.4 illustre la définition de la monade de pré-condition la plus faible hybride correspondant au couple interpréteur abstrait et monade de pré-condition la plus faible définie dans le chapitre 4.3. Une monade hybride de pré-condition la plus faible calcule non seulement une pré-condition, mais aussi, dans le même temps, une interprétation abstraite. Une telle monade hybride produit alors des preuves d'obligations aidées d'invariants provenant de l'interprétation abstraite embarquée. Le théorème de correction de cette construction hybride est détaillé et expliqué dans le chapitre 4.5, puis prouvé dans le chapitre 4.6. Notre travail est finalement généralisé dans le chapitre 4.7, en considérant un interpréteur abstrait comme un transformateur de monades de pré-condition la plus faible.

Chapitre 5. Après avoir étudié comment le système de type de F⋆ pouvait aider à produire des analyses statiques (chapitre 3) et vice-versa (chapitre 4), nous étudions dans le chapitre 5 une forme d'analyse statique assez différente, en étudiant la manière dont les types dépendants et le système d'effets de F* pouvait aider à l'implémention d'une librairie permettant la vérification de politiques de flux d'information (IFC). Notre travail est largement inspiré de Labeled Input Output (LIO) [Ste+11], une librairie Haskell monadique, qui permet de décrire et mettre en oeuvre des politiques d'IFC dynamiquement. Notre librairie, LIO*, permet de vérifier de telles politiques de manière très flexible : une politique d'IFC peut être vérifiée sur un programme de manière statique, dynamique ou une combinaison des deux. Le programmeur peut choisir de vérifier statiquement une politique d'IFC pour éliminer tout coût à l'exécution et garantir un certain comportement pour un composant logiciel critique, tout en choisissant la simplicité d'une vérification dynamique pour un autre. Notre librairie se comporte comme une couche logicielle au dessus de Low* [Pro+17] (un langage dédié dans F* pour écrire des programmes bas niveau, et qui profite d'une extraction vers C grâce à l'outil KreMLin [Pro+17]) : ainsi, il est possible d'écrire des programmes profitant à la fois de notre librairie d'IFC et d'une extraction bas niveau vers C.

En premier lieu, dans le chapitre 5.4, nous présentons GLIO*, une librairie d'IFC entièrement statique qui ne fait qu'ajouter des vérifications statiques, et qui disparaît entièrement[5] à la compilation ou à l'extraction. Ensuite, nous présentons DLIO* dans le chapitre 5.5, qui n'est en réalité qu'une sur-couche de GLIO*, consistant à ajouter des vérifications dynamiques à l'exécution : DLIO* n'est en réalité qu'un client comme les autres de notre librairie statique GLIO*. Enfin, nous abordons dans le chapitre 5.7 le problème de la non-interférence, une propriété fondamentale pour un système d'IFC. Nous étudions une manière de générer et de prouver automatiquement des théorèmes de non-interférence étant donné un client particulier de notre librairie.

[5] En usant d'optimisations et de l'outil KreMLin, les modules constituant notre librairies se retrouvent traduits en des fichiers C vides de toute ligne de code.

CHAPTER 1

Introduction

Our daily life is getting more and more impregnated with interconnected software of all kinds. We chat with our friends, family and colleagues through messaging apps, plan our holidays with Google Maps, share the last meal we cooked on Instragram, entertain ourselves on Netflix or YouTube, manage our money from our smartphone, buy more and more online... Pieces of software are just like virtual Lego bricks; it is *in appearance* easy and cheap to compose them together to grow more and more complex systems. As a result, nowadays the slightest piece of software (e.g. a website or an application) quickly becomes a monster of software dependencies, yielding an impressive complexity. For instance, checking the balance of your account from the pretty and snappy user interface displayed by your banking app yields utterly complex pieces of software, among which we probably find some pieces of old legacy COBOL programs. Among all human-built artifacts, software ones are probably the most complex.

> *Language is the raw material of software engineering, rather as water is the raw material for hydraulic engineering. The difference is that water is rather well understood by physical science; but software –as a raw material– is still not scientifically understood. Nevertheless our software engineers have filled the world with software at enormous speed.* — Robin Milner

One slight misbehavior happening in one of the components of a piece of software may be enough to cause a *bug*. A bug is an incorrect or unexpected software behavior, and obviously, it can cause a variety of annoyances. An example is Toyota's unintended acceleration [Koo14]: it shows that trivial programming errors can lead to severe fatalities. On a different note, a bug in an Etherum contract led to about $50M worth of cryptocurrency vanishing [Fou]. The US National Institute of Standards and Technology (NIST) estimates the cost of software bugs to around $59.5 billions each year [Pla02]. In view of all the problems that defective software can cause, software *quality* is of uppermost importance.

Software is run on computers, which understand machine code. Despite the relative coolness of writing raw machine code, it is neither practical nor productive. As Harold Abelson wrote, "programs must be written for people to read, and only incidentally for machines to execute". Machine code is far from being pleasant to read. Just like good and fresh ingredients sublimate a great dish, good programming languages are essential to software quality. There are hundreds of such languages in the wild: the quest for *The Best Programming Language* is still ongoing. To complete this quest, a first step would be to choose a metric to judge what a good programming language is, and what a bad one is. Seeking for the best quality of software, we are interested in this thesis in programming languages that help the programmer to altogether eliminate bugs. We are interested in languages equipped with strong type systems, implementing features such as *dependent types* or *refinement types*.

1.1 Types and Precision

The first commercially available language was Fortran and already had types in the sense that it had a static distinction between e.g. integers and floating-point numbers. The word type was popularized later on with the Algol programming language. At the time of Fortran and Algol, the available types existed by necessity: the integer 42 and the floating-point number 42.0 are represented by different sequences of bits. Such types are qualified of primitive: they suppose a certain bit-level representation. In the case of such primitive types, typing a value as an "integer" or a "floating-point number" (i) ensures a correct machine interpretation, and (ii) helps the compiler to decide how to lay out that value in memory. Types help the programmer to avoid certain undefined behaviors. For instance, the arithmetic addition is expected to be fed with two numbers; if one was to feed a string and an array to the arithmetic addition operator, one would yield an undefined (potentially dangerous) behavior.

Types can serve higher-level purposes. Modern programming languages such as Haskell [Pey07] or languages of the ML family allow much more expressive types, for example user defined custom types (i.e. inductive types) and function types. While primitive types help the compiler, combining them in compound types helps the programmer.

Verifying that an expression is typed correctly can either happen before or after the execution of a program. A *static type system* reads the source code of a program before its execution and tries to *type-check* it. When static type-checking succeeds, then the program is supposed not to hit type-related issues at run-time. By contrast, a *dynamic type system* checks if a value is typed correctly just before it is actually used in the program, at runtime.

Specification precision. The range of bugs and unexpected runtime behaviors one can avoid with a static type system depends on the level of expressiveness it offers. To get some intuition about which kind of expressiveness a type system can offer, let us consider the following function `sum`. It is a recursive function that computes the sum of integers from 0 to its input `n`:

```
let rec sum n = if n≤0 then 0 else n+sum(n-1)
```

This behavior of the function `sum` can be described more or less precisely. Let us exercise and give `sum` a few *specifications*, ranging from very weak and imprecise ones to very strong and precise ones:

(i) `sum` is a program that takes an input and produces an output;

(ii) `sum` maps an integer to another integer;

(iii) `sum` maps an integer to a non-negative integer;

(iv) `sum` maps `n` an integer to `m` an integer greater than `n`;

(v) sum maps n to $\frac{\max(n,0)^2+\max(n,0)}{2}$.

Figure 1.2 illustrates what it takes for a function to satisfy specifications (iii), (iv) and (v). Visually, it is clear that specification (iii) has a different nature from specification (iv) or (v). Indeed, the specification (iii) restricts the output values of sum independently from its input; by contrast, (iv) and (v) state specifications *taking into account* the input of sum. This notion of dependencies is important: there is a gap of expressiveness between (iii) and (iv)/(v).

Type expressiveness. These various specifications can all be encoded as types, but not all type systems can encode these specifications. Figure 1.1 gives some examples of programming languages whose type system is able to handle these different specifications. Not all type systems are equal. The specification (iii) is easy to encode as type: it is a function that maps integers to positive numbers. Hence, if one defines $\mathbb{N}$ a type that represents positive numbers, the specification is implemented exactly as the *arrow type* $\mathbb{Z} \to \mathbb{N}$. Defining such a type $\mathbb{N}$ is easy. A natural number is either zero or the successor of another natural number, leading to the following *algebraic data type* definition:

```
type ℕ = | 0 | Succ: ℕ→ℕ
```

In comparison with specification (iii) for which we defined $\mathbb{N}$, specification (iv) requires a type $\mathbb{Z}_{ge}$ that depends on a value. More commonly, type systems allow for *polymorphism*, that is, types that are indexed by other types (e.g. lists). By contrast, here we are looking for a type indexed by an integer. For instance, the type $\mathbb{Z}_{ge}\ 4$ is inhabited by every integer greater or equal to 4. Types indexed by values are called *dependent types*. Similarly, given a simple arrow type $\tau \to \beta$, the type β cannot be indexed by input values of type τ. Instead, a dependent arrow type can be in the form $x{:}\tau \to \beta\ x$, and express a dependency. Given an integer-indexed type $\mathbb{Z}_{ge}$ exists, the specification (iv) can be encoded as the dependent arrow type $n{:}\mathbb{Z} \to \mathbb{Z}_{ge}\ n$.

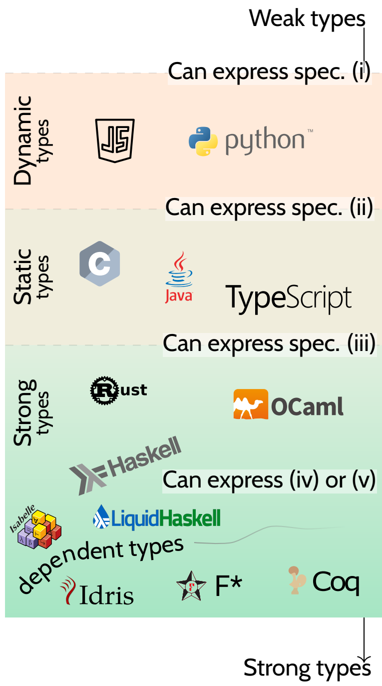

Fig. 1.1: Various programming languages and the expressiveness of their type system.

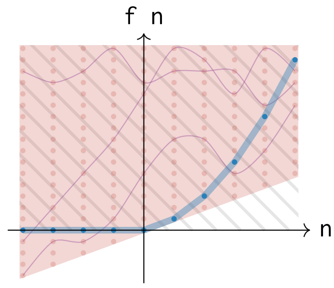

Fig. 1.2: The specification (iii) is represented by the hatched area, the specification (iv) by the red area, and (v) by the blue line. There exists a lot of functions that stick to specification (iv) for instance: the violet lines give some examples of such functions. By contrast, specification (iv) is much more restrictive.

1.2 Type Expressiveness Versus Ease of Use

As a type system gets stronger, its procedure for type-checking becomes undecidable. For example, the type-checking procedure of a type system that checks for termination is trivially undecidable by reduction to the halting problem [Chu36; Tur37]. In consequence, writing a program with such expressive types requires the programmer to assist the type system by supplying hints, annotations or even proofs to the type system.

Our work is rooted in the observation that programming languages that offer a great type expressiveness suffer from automation issues. One of the most well-known programming languages equipped with dependent types is Coq [The04]. Coq is not aimed at general-purpose programming; it is rather a proof assistant, that is, a system that allows to formally state proofs and to prove them interactively. As such, Coq is primarily designed for specifications and proofs to be written in a precise way. Writing programs with specifications as type in Coq often yields great amount of proof obligations, resulting in a lot of proof effort.

Liquid Haskell [Vaz+14] is somehow the opposite approach: it supplements the well-known general-purpose programming language Haskell with more expressive types. It enables Haskell types to be refined with restricted (QF-UFLIA [BST+10]) logical predicates with a great degree of automation. It was extended to properties about arbitrary Haskell functions [Vaz+18], turning Liquid Haskell into a theorem prover.

By contrast to Coq, Idris [Bra13] or F* [Swa+16] are *general-purpose* programming languages equipped with dependent types. Equipped with dependent types and built-in SMT solver facilities, F* provides both an OCaml-like experience and proof assistant capacities. Its type system features both dependent and refinement types, weakest-precondition calculi and monadic effects. It recently shone with the Project Everest [EVEREST] which delivered a series of formally verified, high-performance, cryptographic libraries: HACL* [Zin+17], ValeCrypt [Bon+17] and EverCrypt [Pro+20]; these are for instance used and deployed in Mozilla Firefox. While F* can always resort to proof scripts similar to Coq ones, most proof obligations in F* are automatically discharged by the SMT solver Z3 [DB08]. Even if using SMT solvers can help lower the amount of proofs a programmer shall write to verify that a function matches a specification, it does not help type inference.

Strong typing is not the only approach to formal verification. Most interesting programs are way too complex to be executed and tested thoroughly for every possible input: *abstract interpretation* [CC77] is a brilliant answer to this problem. Instead of attempting at running programs in the –infinite and rough– concrete world to capture properties, the idea is to interpret programs in an abstract world carefully chosen to enjoy pleasant properties, e.g. systematic termination. Such abstract interpretations inevitably yield approximations, but in turn allow for automatic discovery of properties in finite time. Our work aims precisely at better type inference in the settings of strong type systems. In the case of dependent types, type inference amounts to automatic inference of program properties. The thesis defended in this dissertation is the following:

> “*Static analysis –and particularly abstract interpretation– and type systems equipped with dependent types are complementary and can learn from each other.*”

1.3 Contributions and Structure of the Document

The first chapter (Chapter 2) is introductory, and presents how to program in a proof-oriented style with the F⋆ programming language. After a short tutorial to F⋆ basics, we explore the underlying concepts behind *effects* by describing the concepts of computation, specification and indexed monads. The *effect* system is one of the most important –and distinctive– features of F⋆: after looking at their foundations, the chapter ends with an overview of the common use-cases of effects in F⋆.

In Chapter 3, we are interested in abstract interpretation, a theory of sound approximation which is notably used to certify that a software respects certain properties. While abstract interpretation algorithms provide sound approximations, an implementation of abstract interpreter might diverge slightly from these algorithms. There exists provably sound implementations of abstract interpreters. They are mostly written in Coq and yield the best guarantees of soundness, at the cost of *proof scripts*, which are very difficult to understand for those who are not Coq experts. Our work presents a verified sound abstract interpreter implemented in F⋆ with very few manual and explicit proofs. As a result, we are able to fit the entire source code of our interpreter in the chapter, gaining an order of magnitude in terms of amount of proofs required, compared to similar works.

Chapter 4 presents a methodology that gives a way of turning an abstract interpreter into a weakest-precondition monad transformer. The idea of this work is to exploit the expressive power of specification monads that implement a weakest-precondition calculus to inject a property inference mechanism derived from an abstract interpreter. As a result, we get a *hybrid* weakest-precondition calculus that uses abstract interpretation to lighten its computed proof obligations a user shall discharge. As supplementary material, we provide an instance of our methodology, that is a *hybrid* weakest-precondition with a partial mechanized proof of soundness.

Chapter 5 changes of scope and focuses on the verification of *Information Flow Control* (IFC) policies of F⋆ programs. It implements an IFC system as a library whose originality lies in the fact it enables different shades of verification, from fully static to fully dynamic, according to the need of the programmer. This chapter describes another use case of static analysis embedded in a type system: our library infers IFC-related properties about programs. The F⋆ clients of our library can also be extracted to C code when they are written in the Low⋆ subset, which enjoys compilation to C.

To conclude this dissertation, Chapter 6 summarizes our contributions and discusses possible extensions to our work.

Notes about Chapters 3, 4, and 5

Online material Chapter 3, 4, and 5 have their companion F⋆ implementations available as supplementary materials at the

URL `https://lucas.franceschino.fr/phd-thesis/`.

Contributions The work presented in Chapter 3 has been accepted for publication at the 28th Symposium on Static Analysis (SAS21) [FPT21]. Chapter 5 relates to a collaborative work with Jean-Joseph Marty, Jean-Pierre Talpin and Niki Vazou and was the suject of a pre-publication on ArXiV [Mar+20]. However, Chapter 5 presents an entirely revised version of this work. I'm the single contributor of the formal developments of Chapter 3 and 4. I authored the F* implementation of the library presented in Chapter 5, and designed most of its meta-programming procedure.

CHAPTER 2

Verified Programming and F*

Contents

F* is a general purpose functional programming language. It is aimed at verified programming: it features dependent types and refinement types, allowing for proving properties of programs. This chapter assumes the reader is familiar with a functional programming language such as OCaml, with which F* shares a similar syntax. It presents how to write verified programs in F*, and details some of its features and foundations.

Classically, most of the code that one writes corresponds to the different steps necessary to *compute* a result, in order to solve a given problem. In verified programming, one also writes *specifications* and *proofs* which have no impact on computations. The sole aim of such a computationally irrelevant code is to *verify* properties about portions of programs.

Section 2.1 goes through various simple programs to present how verified programs can be written in F*. Then Section 2.2 introduces the notion of computation and specification monads, and finishes with a presentation of F* Dijkstra monads. The latter play an important role in one of the major features of F*: *effects*. This feature gives F* a modular means to verify programs with a wide spectrum of side effects. Section 2.3 presents what effects are made of, and highlights a selection of interesting use-cases in Sections 2.3.2 and 2.3.3.

2.1 Writing and Proving Functional Programs

This section starts with a short tutorial to functional programming in F*. We exhibit a selection of features and syntaxes in use in the rest of this document.

2.1.1 Refinement Types

From the point of view of an F* user, refinement types is the most important feature of F*: they allow for simple and flexible, yet powerful specifications. The syntax $\mathtt{x}:\tau\{\phi\}$ denotes the refinement of the base type τ by the formula ϕ that might refer to the variable $\mathtt{x}$. A simple example of type refinement is the definition of the type for natural numbers. Given $\mathbb{Z}$ the type of relative numbers, $\mathbb{N}$ = $\mathtt{n}:\mathbb{Z}\{\mathtt{n}\geq 0\}$ is the type of natural numbers. Fig. 2.1 gives some more examples.

```
let even  = n:ℤ{n % 2 == 0}
let odd   = n:ℤ{n % 2 == 1}
let empty = _:unit{⊥}
```

Fig. 2.1: Refinement types examples.The type `empty` is inhabited by nothing: it is isomorphic to type ⊥.

Below, we give the F* definition of `fact`, that computes a factorial. The **`let rec`** denotes a recursive top-level declaration. The type of `fact` is `n:ℕ → r:pos{r≥n}`: such a type is called an *arrow type*. Moreover, the return type `r:pos{r≥n}` refers not only to `r`, but is also parameterised by `n`: we thus call it a *dependent type*. Here, the refinement types act as pre- and post-conditions: given `n` a non-negative integer, the function returns a strictly positive number greater or equal than `n`. Figure 2.2 illustrates this refinement type. The declaration `pos` is a *type synonym*: we do not introduce any type constructor, we just introduce a name for the refinement `n:ℤ{n>0}`.

```
type pos = n:ℤ{n>0}
let rec fact (n: ℕ): r:pos{r ≥ n}
= if n = 0 then ❶ 1
           else ❷ multiply (fact (n-1)) n
```
(2.1)

Let us review manually why `fact` typechecks. For any natural number `n`, `fact n` should be of type `r:pos{r≥n}`. *Eliminating* the refinement type, the proof obligation becomes `(if n=0 then 1 else ❷) ≥ n`. In other words, the proof obligation is the conjunction of `n=0` $\implies$ `1≥n` and `n≠0` $\implies$ `❷≥n`. The left part of the conjunction is trivial. Let us look at the formula `❷ ≥ n` under the hypothesis `n≠0`:

- `fact` expects `n-1` to be of type ℕ. By elimination of the refinement held in ℕ, this expectation amounts to the proof obligation `n - 1 ≥ 0`. The conjunction of our hypothesis `n ≠ 0` with the elimination of the refinement `n:ℕ` gives us our objective `n-1≥0`.
- We can now use our recursive call `fact (n-1)`, that has the type `r:pos{r ≥ n-1}`. This latter type is a subtype of `pos`: eliminating this refinement type, we get `fact (n-1)≥1`. `❷ ≥ n` is now trivial for the SMT solver: the multiplication `fact (n-1)` by `n` is greater or equal to `1×n`.

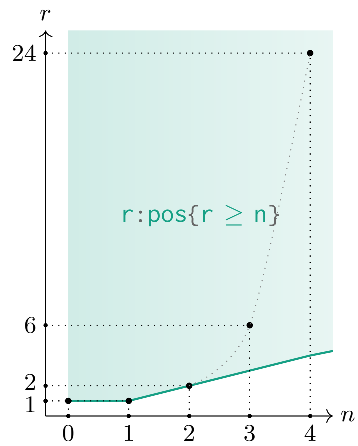

Fig. 2.2: The green shape represents the dependent type `r:pos{r ≥ n}`. The 5 points corresponds to the 5 first values of the factorial function. The type `n:ℕ→r:pos{r ≥ n}` captures only a little of the behavior of the factorial function.

2.1.2 Inductive Types

Without surprise, F* allows the user to define custom types. Below, we define two inductive types: one for simple lists (`list`), and one for lists with specific lengths (`vector`). The inductive type `list` is indexed over τ a type: the type of `list` is thus **`Type`** → **`Type`**. However, in F*, a type can be indexed by any sort of values, not only by types. An example of such a type is `vector`, whose type is **`Type`** → ℕ → **`Type`**.

```
type list (a: Type): Type
  = | Cons: hd:a → tl:list a → list a
    | Nil: list a
type vector (a: Type): ℕ → Type
  = | VCons: #n:ℕ → hd:a → tl:vector a n → vector a (n+1)
    | VNil: vector a 0
```

Accessors, implicit types and discriminators For each constructor, F* produces one discriminator and a set of accessors. In the case of the constructor `Cons`: F* produces the accessors `Cons?.hd` and `Cons?.tl`, along with a discriminator `Cons?`. Their type are given below, and are arrow types. In a function type, the syntax `#x:t` denotes an implicit argument named `x` of type `t`. A very common use-case for implicit arguments is polymorphism. A polymorphic function takes one or multiple type(s) as parameter(s). Such parameters are redundant: they can generally be inferred automatically looking at other parameters. In the declaration `implicit`, `Cons 42 Nil` is of type `list ℤ`, and the parameter `#a` is inferred automatically as ℤ. In opposition, definition `explicit` fixes the implicit type `a` manually, to set it as the refined type ℕ.

```
Cons?     : #a:Type → l:list a → bool
Cons?.hd  : #a:Type → l:list a{Cons? l} →      a
Cons?.tl  : #a:Type → l:list a{Cons? l} → list a

let implicit = Cons?.hd    (Cons 42 Nil)
let explicit = Cons?.hd #ℕ (Cons 42 Nil)
```

List of fixed lengths The inductive type `vector` is designed specifically to keep track of one specific property: the length of lists. This property is established by the constructors themselves: a value of type `vector ℤ n` is, by construction, a list of `n` elements.

However, in F*, lists with specific lengths might be defined in another, easier way: with refinements. Below, `len` maps lists to their lengths. The type synonym `vector'` takes advantage of this definition and provides an alternative type for fixed-length lists.

```
let rec len (l: list τ): ℕ
  = match l with | Cons _ tl → 1 + len tl
                 | Nil → 0
type vector' (a: Type) (n: ℕ) = l: list a {len l == n}
```

This second definition `vector'` is simpler, and any function that operates on `list τ` will also operate on `vector' τ n` for any `n`. If `x` *inhabits*[1] the type `vector' τ n`, `x` is a list for which a proof that `len l == n` exists. Refinement types are very flexible and simple to use: `vector'` is the standard way to define length-constrained lists in F*.

[1] When a value `x` is of some type τ, we say `x` inhabits the type τ.

2.1.3 Inductive Proofs

We begin by defining `nth`, a function that dereferences the nth element of a list. The index argument `i` is refined to be a valid index for the list `l`. `i` is a witness that there exists at least one valid index for `l`, thus `l` is not empty. In consequence, we know that `l` was constructed with `Cons`. ❶ destructs directly the list into `hd` its head and `tl` its tail, without pattern matching. `nth` expects the subtraction ❷ to be a valid index for `tl`. (i) By ❸ and `i` being a natural number, (ii) by the refinement of `i`, and

(iii) by the unrolling of function len, we have (i) i > 1, (ii) i < len l and (iii) len tl = len l - 1. The SMT solver deduces i - 1 < len tl, which can be introduced as a refinement so that i is subtyped as j:ℕ{j < len tl}.

```
let rec nth (l: list τ) (i: ℕ {i < len l}): τ
  = let ❶ Cons hd tl = l in
    if ❸ i = 0 then hd else nth tl (❷ i - 1)
```

Most languages, devoid of dependent types, cannot express the refinement nth had. They thus implement nth as a partial function. In our case, the SMT solver is able to automate the inductive proof that nth is a total function.

We now present two functions that assert the membership of an element in a list. The first one, mem_0 (Equation 2.2), is straigthforward: it maps any list and element to a boolean, by unfolding the list recursively.

```
let rec mem₀ (#a:eqtype) (l: list a) (e: a): bool
  = match l with
  | Cons hd tl → hd = e || mem₀ tl e
  | Nil → false
```
(2.2)

The second one, mem (Equation 2.3), does an identical job, but also proves that if mem l x holds, there exists an index i such that nth i equals x. The refinement ❶ acts as a lemma about the function mem here. Such an embedded lemma is called intrinsic. When the list is empty, the lemma is trivial: there exists no index for an empty list, thus false ⟺ (∃ i. nth Nil i == e). Otherwise, the list can be destructed as Cons hd tl. Then, when the head hd equals the element e we are looking for, ∃ i. nth (Cons hd tl) i == e holds: nth (Cons hd tl) 0 equals to e by definition of nth. The line ❷ helps F* by asserting that latter fact as an intermediate lemma. When hd is not equal to e, we can use our hypothesis of recurrence introduced by the recursive call mem tl e, that we reformulate with ❸.

```
let rec mem (#a:eqtype) (l: list a) (e: a)
  : (r:bool{❶ r ⟺ (∃ i. nth l i == e)})
  = match l with
  | Cons hd tl → ❷ assert (nth l 0 == hd);
                 ❸ assert (∀(i:ℕ{i < len tl}).
                          nth (Cons hd tl) (i + 1)
                          == nth tl i);
                 ❹ hd = e || mem tl e
  | Nil → false
```
(2.3)

Note that here, the assertions ❷ and ❸ are mandatory. If one of them is omitted, F* fails type-checking. F* fails at subtyping the boolean we return at ❹ as a boolean refined with the predicate ❶. In this case, these assertions are in fact intermediate lemmas about our function nth. To find such missing assertions, the F* programmer can play with admit expressions, that admit a given statement holds.

Type universes Until now, we wrote **Type** (or `eqtype` above) to denote the "type of types". However we sometimes need more precision (e.g. Section 4.2.2 or 5.2.2). In F⋆, types are organized along a sequence of non-cumulative type universes [Mou+15]. The first type universe, denoted **Type** `u#0` (and abbreviated $\texttt{Type}_0$), is inhabited by "ordinary" types, such as $\mathbb{Z}$ or `list` $\mathbb{Z}$. Then, the first universe of type $\texttt{Type}_0$ inhabits the second universe **Type** `u#1`, which itself inhabits **Type** `u#2`, etc. However, type universes are not cumulative: for instance, **Type** `u#0` doesn't inhabit **Type** `u#2`. The F⋆ syntax `x <:` τ denotes type ascriptions. Below, we use type ascription abusively to demonstrate how type universes nest in F⋆.

```
Type u#0 <: (Type u#1 <: (Type u#1 <: (Type u#2 <: ...)))
```

The **Type** notation, with no universe information, denotes an arbitrary universe of types. F⋆ has an inference mechanism for universes. Thus, universes are, most of the time, invisible and left implicit. F⋆ also has universe polymorphism: in Section 2.1.2, we claim that `list` is of type **Type** → **Type**. Showing explicitly type universe, `list` has the (universe polymorphic) type **Type** `u#n` → **Type** `u#n`. Universe expressions obey the following grammar: (i) natural number literals (i.e. `u#3`), (ii) universe variables (i.e. `u#foo`), (iii) addition between literal constants and universe variables (i.e. `u#(foo + 4)`), (iv) or the maximum of any universe expressions (i.e. `u#(max (foo + 4) bar)`). A universe polymorphic value can be monomorphized: for instance, `list u#5` has the non-universe-polymorphic type **Type** `u#5` → **Type** `u#5`.

An inductive type with a constructor that holds a value of type **Type** `u#n` will live in the type universe of rank at least `n+1`. Below, we give an example of such a constructor by defining heterogeneous lists. The constructor **HCons** takes as first argument (❶) a type, and a value of that type as second argument (❷). The definition $\texttt{ex}_1$ is a heterogeneous list of values ($\mathbb{N}$ and ⊤). Its type inhabits **Type** `u#1`; in opposition, the type of $\texttt{ex}_0$ inhabits $\texttt{Type}_0$. The definition $\texttt{ex}_2$ fails[2] to be typechecked: F⋆ handles universe polymorphism only on top-level declarations.

```
noeq type hlist: Type u#(n+1) =
  | HCons: ❶ t:Type u#n → ❷ hd:t → tl:hlist → hlist
  | HNil: hlist
let ex0: hlist u#0 = HCons string "Hello!" (HCons ℤ 4 HNil)
let ex1: hlist u#1 = HCons Type0 ℕ (HCons Type0 ⊤ HNil)
[@@expect_failure] let ex2 = HCons Type0 ℕ (HCons ℕ 1 HNil)
```

Coming back to our definitions of `mem` and $\texttt{mem}_0$, the type `eqtype` is specifically a subset of $\texttt{Type}_0$. A type `t` inhabits `eqtype` if there exists a decidable equality[3] Such an equality is automatically generated when possible by F⋆ when defining an inductive type. This generation can be disabled with **noeq** (as `hlist` does above).

Extrinsic lemmas While the refinement on `mem`'s outcome acts as an intrinsic lemma, one can also state detached (or *extrinsic*) lemmas. Below,

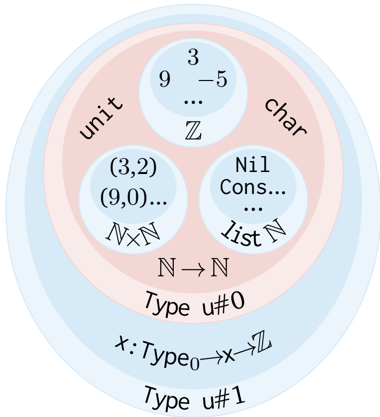

Fig. 2.3: Type universe illustration.

[2] In F⋆, declarations, inner let bindings, binders in arrow types, and record fields can be decorated with attributes. An attribute is an arbitrary F⋆ term. The listing below decorates $\texttt{ex}_2$ with `expect_failure`, causing F⋆ to expect the typechecking of $\texttt{ex}_2$ to fail. Other attribute of interest includes `tcnorm` that requests the typechecker to normalize a declaration, or `plugin`, that marks a declaration for native plugin compilation.

[3] That is, a function of type `x:t → y:t → r:bool{r ⟺ x == y}`: a computable boolean equality that respects propositional equality. More precisely, `eqtype` is a refinement over the universe of type $\texttt{Type}_0$: `a:Type0{hasEq a}`. The predicate `hasEq` is axomatized, and mirrors the types for which F⋆ was able to generate decidable equalities.

`mem_eq_lemma` is a lemma that demonstrates the equality between our two functions `mem` and $\texttt{mem}_0$. It is a very simple proof by induction.

$$\begin{array}{l}\texttt{let rec mem\_eq\_lemma (\#a: eqtype) (l: list a) (x: a)} \\ \quad \texttt{: Lemma (mem}_0 \texttt{ l x == mem l x)} \\ \quad \texttt{= match l with | Cons hd tl} \rightarrow \texttt{mem\_eq\_lemma tl x} \\ \qquad\qquad\qquad\quad\; \texttt{| Nil} \rightarrow \texttt{()}\end{array} \tag{2.4}$$

The reader might wonder why, in the case where `l` is `Nil`, we simply wrote `()`, which denotes the only inhabitant of the type `unit`.

In Section 2.1.2, we explain that a value of type `l:list{len l == 3}` is a list for which there exists a proof that `len l == 3`. Similarly, a value of type $\texttt{\_:unit\{mem}_0\texttt{ Nil x == mem Nil x\}}$ is a value for which there exists a proof that $\texttt{mem}_0\texttt{ Nil x == mem Nil x}$. The point is that F⋆ encodes proofs (or lemmas) as particular refinement of the uninformative type `unit`.

Coming back to our proof when `l` is the empty list, the sub-goal we are trying to prove is $\texttt{mem}_0\texttt{ Nil x == mem Nil x}$. When we simply write `()`, F⋆ asks the SMT solver whether the value `()` is of type $\texttt{\_:unit\{mem}_0\texttt{ Nil x == mem Nil x\}}$. This amounts to questioning whether $\texttt{\_:unit\{mem}_0\texttt{ Nil x == mem Nil x\}}$ is a subtype of `unit` (or $\texttt{\_:unit\{}\top\texttt{\}}$), which itself amounts to proving the following statement, that is proven automatically by the SMT solver:

$$\top \implies \texttt{mem}_0\texttt{ Nil x == mem Nil x}$$

With `l` non-empty, we use our induction hypothesis `mem_eq_lemma tl x` which has type $\texttt{\_:unit\{mem}_0\texttt{ tl x == mem tl x\}}$. To solve the proof goal, F⋆ tries to subtype `mem_eq_lemma tl x` as $\texttt{\_:unit\{mem}_0\texttt{ l x == mem l x\}}$. By elimination of these refinements, the proof obligation becomes:

$$(\texttt{mem}_0\texttt{ tl x == mem tl x}) \implies (\texttt{mem}_0\texttt{ l x == mem l x})$$

Unfolding $\texttt{mem}_0$ and `mem`, the SMT solver discharges the proof obligation.

The last bit of syntax which we left unexplained is the syntax `Lemma`. Until now, every arrow type we gave was of the form $\tau_0 \rightarrow \tau_1 \rightarrow \ldots \rightarrow \beta$, with τ_i and β being any type. The syntax $\tau_0 \rightarrow \tau_1 \rightarrow \ldots \rightarrow \beta$ is actually a syntactic sugar for $\tau_0 \rightarrow \tau_1 \rightarrow \ldots \rightarrow \texttt{Tot}\ \beta$. `Tot` indicates pure and total computations. In F⋆, arrow types are[4] of the shape $\tau_0 \rightarrow \tau_1 \rightarrow \ldots \rightarrow \texttt{E}\ \beta$ where `E` is an *effect*. `Lemma` is simply an effect.

This section presented what F⋆ is made of, and how it can be used. We will now dive into its internals by discussing effects and Dijkstra monads.

2.2 Dijkstra Monads and Effects

As the previous section underlined, one of the strengths of F⋆ is refined types and the way subtyping is (mostly) decided by an SMT solver. Another key feature of F⋆ is its built-in *effect system*, allowing computations to perform side-effects and the programmer to reason precisely about them. Before

[4]More specifically, an arrow type of arity 1 is of the shape $\texttt{x:}\tau \rightarrow \texttt{E}\ (\texttt{y:}\beta\ \texttt{x})\ \texttt{e}_1\ \ldots\ \texttt{e}_n$, with τ:`Type` and $\texttt{b:}\tau \rightarrow$ `Type`. The n arguments $\texttt{e}_1\ \ldots\ \texttt{e}_n$ are *effect indexes* (see 2.3). Arrow type of bigger arity can be derived by nesting. Three alternative syntaxes exist for the *binder* `x`:

- $\texttt{\#x:}\tau$ denotes an implicit binder;
- $\texttt{(\#[tau] x:}\tau\texttt{)}$ denotes an implicit binder resolved by the ad-hoc tactic `tau`;
- `{| tc ... |}` denotes a typeclass constraint (which are used a lot in Chapter 3).

diving into F* effects, this section provides some background about monads, which are the basis of effects.

Real-world programming inherently involves a great number of side effects that can take many different forms. Non-determinism, interaction with the external world, exceptions or stateful computations are all side-effects. Most programming languages handle such side-effects via dedicated and ad hoc syntax. For instance, JavaScript supports exceptions via a dedicated mechanism and syntax **throw**, **try** and **catch**. Such side-effects or *language feature* can however be abstracted in a uniform manner thanks to *monads*[Mog91; PW93].

2.2.1 Computational Monads

A monad is an algebraic structure consisting in: (i) a *representation*, in the form of a type `M: Type→Type`; (ii) a *return* operation $\mathtt{ret}_M : \tau \to \mathtt{M}\ \tau$; (iii) a *bind* operation $(\gg=_M) : \mathtt{M}\ \tau \to (\tau \to \mathtt{M}\ \beta) \to \mathtt{M}\ \beta$. A bind operator defines what the composition of two monadic computations is. The return operator lifts a value as a monadic computation. The bind and return operations are *expected* to obey certain laws: those are presented in Figure 2.4 and explained by the quotation below it.

Left and right identity:

$$\mathtt{ret}_M\ x \gg=_M f \equiv f\ x$$
$$f \gg=_M \mathtt{ret}_M \equiv f$$

Associativiy:

$$(f \gg=_M g) \gg=_M h \equiv f \gg=_M (g \gg=_M h)$$

Fig. 2.4: Monad laws.

> “ Left identity: *The first monad law states that if we take a value, put it in a default context with return and then feed it to a function by using bind, it's the same as just taking the value and applying the function to it.*
> Right identity: *The second law states that if we have a monadic value and we use bind to feed it to return, the result is our original monadic value.*
> Associativity: *The final monad law says that when we have a chain of monadic function applications with bind, it shouldn't matter how they're nested.*
> — Learn You a Haskell ”

In the scope of a purely functional language, every object is just a function, that is, a relation associating each element of a given domain to a single element of its codomain. Thus, as such, features like non-determinism are impossible to bring to pure functions. A great property for pure functions is to have referential transparency [WR10]: a call to a pure function can be replaced by its value without changing the meaning of the program. Monads model *computation*, not *functions*.

Let us be more concrete with an example: the state monad `st`. The state monad represents *computations* that can *read* and *write* values from a store. The word computation is important: a computation is *not* a function (i.e. not a relation). A computation `c` of type `st` $\mathbb{Z}$ that returns its current state is not subject to referential transparency. Indeed, the evaluation of `c` at time t_1 cannot be replaced with a previous evaluation of `c` at time t_0. `c`'s *outcome* depends on its context. The monad `st` *represents* stateful computations.

Figure 2.5 presents a functional and pure implementation for the monad `st`. A stateful computation of type `st` τ can be seen as a function mapping an initial store to a final store and a value τ, whence the type `st`. The operation $\mathtt{return}_{\mathtt{st}}$ `x` injects a pure value `x` in the monad `st`, resulting in a constant store transformer. The bind operation $\mathtt{bind}_{\mathtt{st}}$ `f g` first *computes* `f` and feeds `f`'s final store to `g`. The `read` and `write` are the *actions* of the monad `st`: they are the interface to the two features offered by the state monad. `read` and `write` are *computations* in `st`.

```
type st (a: Type) = s → a × s
let return_st (x: τ): st τ     = λs₀ → x, s₀
let write_st  (x: s): st unit = λ_ → (), x

let bind_st (f: st τ) (g: τ → st β)
            = λs₀ → let x, s₁ = f s₀ in
              g x s₁
let read_st : st s = λs₀ → s₀, s₀
```

Fig. 2.5: Definition of the state monad.

Monads can encode a very wide variety of language features. While such monad-encoded functionalities usually concern computations, we will see that monads manipulating type-level information exist as well, and are of particular interest.

2.2.2 An Interlude: Hoare Logic and Weakest-Preconditions

As the reader might suspect, the kind of type-level monads we are particularly interested in involves weakest-preconditions. Thus, before discussing type-level monads, let us look at the particular class of denotational semantics that are weakest precondition calculi, and at Hoare logic.

Note: For simplicity in this subsection we let free variables of pre- and post-conditions implicitly refer to variables from either initial or final context from the program in stake.

The Hoare triple $\{\,P\,\}\ f\ \{\,Q\,\}$ is a logical statement that holds when, for an initial context where the predicate P holds, the context after evaluating f satisfies the predicate Q. As an example, let's consider the triple $\{\,X\,\}\ \mathit{fact}\ \{\,r > 100\,\}$, with fact the imperative program from Figure 2.6 that computes the factorial of the number stored at variable n into an other variable r. We are looking for X a suitable pre-condition so that evaluating fact leads to a context in which r is greater than 100. Since $5!$ equals 120, a suitable precondition X is `n=5`. This precondition is far from being unique: after all, $6!$ or $10!$ are also greater than 100. More specifically, any predicate that implies the factorial of n to be greater than 100 is a valid precondition. Defined below, $\mathbb{X}$ is the set of preconditions so that $X \in \mathbb{X} \implies \{\,X\,\}\ \mathit{fact}\ \{\,r > 100\,\}$.

```
i = 0; r = 1;
while (i < n)
  { i = i+1; r = r*i; }
```

Fig. 2.6: fact program.

$$\mathbb{X} = \{P \mid \{\,P\,\}\ \mathit{fact}\ \{\,r > 100\,\}\}$$

Logical propositions are partially ordered by implication. Note that $\mathbb{X}$ is not empty: it trivially contains $\perp$. The set $\mathbb{X}$ contains the preconditions that are sufficient to prove the post-condition `r > 100`. While the precondition `n = 42` is sufficient, it is clearly not necessary. Hence, among the preconditions in $\mathbb{X}$, we are looking for the loosest possible, the most permissible one. Such a pre-condition is called a weakest-precondition.

As we just saw, Hoare logic allows one to verify whether a post-condition holds after executing a program given a certain pre-condition. A weakest-precondition calculus is a set of computable functions that, given a code fragment f and a post-condition P, computes a weakest-precondition X such that $\{\,X\,\}\ f\ \{\,P\,\}$.

2.2.3 Specification Monads

As discussed previously, monads usually represent computationally relevant behaviors: for instance, throwing an exception changes the control flow and semantics of a program. By contrast to such computational monads, *specification monad* are purely producing type-level information, leaving aside any concrete outcome.

A *specification monad* is defined as a monad whose representation type is inhabited by non-informative values. The reader might then wonder how such monads, computing only type-level information, are any different from more standard and straightforward static analysis techniques. Indeed, evaluating a type-level only computation is basically the same operation as performing a static analysis. However, specification monads and computational monads can be arranged together as *indexed monads*. Such monads enjoy the benefits of both specification and computation monads.

Below, Section 2.2.4 defines an example of weakest-precondition monad, that is, a certain sort of specification monad. Section 2.2.5 presents what *indexed monad* are on a simple example, so that Section 2.2.6 defines a monad indexed by a weakest-precondition monad.

2.2.4 Weakest-Precondition Monad

A weakest-precondition monad is a specification monad that computes weakest-preconditions. As an example, we define such a monad for stateful computations. We define the type `st` of state, which is inhabited by partial maps from addresses to integers. A pre-condition in this setting maps an initial state to a proposition[5]; a post-condition maps an outcome value and a final state to a proposition. A weakest-precondition transforms post-conditions into pre-conditions.

[5] In opposition to booleans, propositions of type $\mathbf{Type}_0$ can represent arbitrary non-decidable logical statements.

```
let state = address:ℤ → option ℤ
let pre = s₀:state → Type₀
let post a = value:a → s₁:state → Type₀
let wp a = post a → pre
```

Let us now write a monad that produces weakest-preconditions: a monad whose representation is `wp`. Note that inhabitants of `wp` τ are continuations. A value x is lifted as a weakest-precondition by writing a continuation: given a post-condition p and an initial state s_0, lifting x does not update the state, thus its precondition is just p fed with x and s_0. Thus, we simply pass x and the initial state to the post-condition. Binding a computation[6] `f` to a computation g is done by feeding g as a post-condition to `f`. In other words, to compute the weakest-precondition of "`f` then g" given a post-condition p, we compute the weakest-precondition of `f` given the precondition required so that g holds given p. Due to the continuation nature of `wp`, our monad somehow computes in a backward fashion.

[6] The word "computation" is to be understood as "an inhabitant of the representation type of the monad in stake". The monad in stake having weakest-precondition as representation, a computation here is a weakest-precondition.

```
let return_w (v:τ): wp τ
  = λ(p: post τ) s₀ → p v s₀
let bind_w (f: wp τ) (g: τ → wp β): wp β
  = λp s₀ → f (λx s₁ → g x p s₁) s₀
```

The `store` operation returns nothing, but performs a side effect. `store address v` respects a post-condition `p:post unit` when `p () s₁` holds, with s_1 an updated state. `read address` requires its initial state to be initialized at `address`.

```
let store_w (address: ℤ) (v: ℤ): wp unit
  = λp s₀ → p () (λi → if i = address then Some v else s₀ i)
let read_w (address: ℤ): wp ℤ
  = λp s₀ → match s₀ address with | Some v → p v s₀
                                  | None → ⊥
```

The actions we introduced ($\mathtt{return_W}$, $\mathtt{bind_W}$, $\mathtt{store_W}$ and $\mathtt{read_W}$), along with the representation type wp form a specification monad of weakest-precondition. We call this monad **W**.

2.2.5 Indexed Monads

As a motivating example, let us consider a state monad which deals with finite stacks of numbers, modeled as lists. Just as the state monad from Figure 2.5, it is tempting to choose a simple representation to define our monad, as we do below with $\mathtt{repr}^{\text{NAIVE}}_{\mathtt{stack}}$. Notice that this definition is extremely similar to the one given in Figure 2.5.

```
type repr^NAIVE_stack (a: Type): Type = list ℤ → a × list ℤ
let return^NAIVE_stack (v: τ): repr^NAIVE_stack τ = λs₀ → v, s₀
let bind^NAIVE_stack (f: repr^NAIVE_stack τ) (g: τ → repr^NAIVE_stack τ): repr^NAIVE_stack τ
  = λs₀ → let r, s₁ = f s₀ in g r s₁
let push^NAIVE_stack (v:ℤ): repr^NAIVE_stack unit = λstack → (), v::stack
[@@expect_failure]
let pop^NAIVE_stack : repr^NAIVE_stack ℤ = λ(❶v::stack) → v, stack
```

Here, the catch is that, while the operation pushing a value on the stack is easy to define, it is not possible to define $\mathtt{pop}^{\text{NAIVE}}_{\mathtt{stack}}$ since the destruction of arbitrary list (at ❶) is a partial operation. The representation type of our monad, $\mathtt{repr}^{\text{NAIVE}}_{\mathtt{stack}}$ is not expressive enough to state whether computations produce stacks that are empty or not.

In consequence, below we define the type $\mathtt{repr}_{\mathtt{stack}}$, which is indexed with two numbers before and after. A stack computation is thus defined as a map from lists of length before to a tuple whose right field is a list of size after.

```
type repr_stack (a: Type) (before: ℕ) (after: ℕ): Type
  = (s₀:list ℤ {before == length s₀})
          → a × (s₁:list ℤ {after == length s₁})
```

This extra expressiveness allows us to quantify the side effects of a stack computation. For example, the type of $\mathtt{return}_{\mathtt{stack}}$ makes sure returning a value in our monadic context leaves the size of the stack unchanged. Similarly, the type of $\mathtt{bind}_{\mathtt{stack}}$ restricts how two computations can be composed, e.g. a computation f that produces an empty stack cannot be composed with g, a computation that expects a non-empty stack.

```
let return_stack #n (v: τ): repr_stack τ n n = λs₀ → v, s₀
let bind_stack (#before_f #after_f #after_g: ℕ)
               (f:      repr_stack τ before_f after_f)
               (g: τ → repr_stack τ after_f  after_g)
               : repr_stack τ before_f after_g
  = λs₀ → let r, s₁ = f s₀ in g r s₁
```

For any n, the action $\mathtt{push}_{\mathtt{stack}}$ is a computation that transports a stack of size n to a stack of size n+1. Now pop is easy to define as a stack computation that expects a non-empty initial stack.

```
let push_stack  (#n: ℕ) (v:ℤ): repr_stack unit n (n+1)
  = λ     stack  → (), v::stack
let pop_stack   (#n: pos)     : repr_stack ℤ n (n-1)
  = λ(v::stack) → v  ,     stack
```

The indexed monad we just defined is limited. Its indexes are two numbers, and only allow to state properties about stack sizes. Instead of indexing a monad simply by a pair of integer, the following section defines an monad indexed by a weakest-precondition monad.

2.2.6 A Computational Monad Indexed By a Weakest-Precondition Specification Monad

Our objective here is to write a computational monad **M** equipped with the weakest-precondition calculus implemented by **W**. A computation in **W** (of type `wp` τ) cannot be executed: **W** gives no computation model, only a specification. **M** should hence implement an actual model of computation. Just as the `st` monad of Figure 2.5, **M** computations are represented as state transformers.

However, its representation $\texttt{repr}_\texttt{M}$ is more complicated. Just like the representation $\texttt{repr}_\texttt{stack}$ of our indexed state monad defined in Section 2.2.5, $\texttt{repr}_\texttt{M}$ is indexed by two values. $\texttt{f:repr}_\texttt{M}\ \mathbb{Z}\ \texttt{w}$ is the representation for a monadic computation producing integers and respecting the weakest-precondition `w`. `f` is a state transformer parameterized by post-conditions.

```
let repr_M (a: Type) (w: wp a)
  = p: post a
  → s0:state {w p s0}
  → r:(a × state) { let v, s1 = r in p v s1 }
```

On a computational level, **M** is really similar to `st` of Figure 2.5. As highlighted below, putting the type and the post-condition aside, the definition $\texttt{return}_\texttt{M}$ and $\texttt{bind}_\texttt{M}$ are very similar to $\texttt{return}_\texttt{st}$ and $\texttt{bind}_\texttt{st}$.

```
let return_M (v:τ): repr_M τ (return_W v)
  = λ_ s → v, s
let bind_M #fW #gW (f: repr_M τ fW) (g: (x:τ) → repr_M β (gW x))
  : repr_M β (bind_W fW gW)
  = λp s0 → let x, s1 = f (λx → gW x p) s0 in
            g x p s1
```

Let us review the type of $\texttt{return}_\texttt{W}$ and $\texttt{bind}_\texttt{W}$. Returning a value $\texttt{v:}\tau$ with $\texttt{return}_\texttt{W}\ \texttt{v}$ is a computation in **M** of type τ indexed by the weakest-precondition $\texttt{return}_\texttt{W}\ \texttt{v}$. The specification is completely delegated to **W**. Consider `f` a computation in **M** of type τ, indexed with the weakest-precondition `fW`, and `g` a continuation from a value `x` of type τ to a computation in **M** of type β indexed by a weakest-precondition `gW x`. Notice that `gW x` is, symetrically to `g` itself, not a weakest-precondition, but a continuation to a weakest precondition[7]. Binding `f` and `g` results in a

[7] Keep in mind that computations in monad **W** and weakest-preconditions are one same thing.

computation in **M** of type β, indexed by the monadic **W** bind of its respective weakest-preconditions.

In our monad **M**, two monadic computations occur at the same time. One in the world of the values (that is, what follows the equal sign in the definitions $\texttt{return}_{\texttt{M}}$ and $\texttt{bind}_{\texttt{M}}$ above), and one in the world of types (that is, computing the index of the computation in stake). This exact mechanism is also used for the last two actions to be defined.

```
let storeM (address: ℤ) (v: ℤ)
  : reprM unit (storeW address v)
  = λp s0 → (), (λi → if i = address then Some v else s0 i)
let readM (address: ℤ)
  : reprM ℤ (readW address)
  = λp s0 → match s0 address with
          | Some v → v, s0
```

We have shown how to construct **M**, a monad indexed by **W** a weakest-precondition monad. The notation of Dijkstra Monads includes monads such as **M**.

2.2.7 Dijkstra Monads

The term of *Dijkstra monad* was first introduced by [Swa+13], from the observation that a weakest-precondition calculus forms a monad. Previous works (Hoare Type Theory [NMB08] and Ynot [Nan+08]) have paved the way up to Dijkstra monad by i.e. exhibiting Hoare monads, that is, monads indexed by pre- and post-conditions.

In the literature, Dijkstra monads refer either to specification monads producing weakest-preconditions [Swa+16], or to computational monads indexed by specification monad producing weakest-preconditions [Swa+13; Ras+21]. These two different definitions are related to the history of how effects have been implemented in F*, so far in three flavors: *primitive*, *dm4free/dm4all* and *layered*. The following Section 2.3 leverages the notion of specification and Dijkstra monads previously highlighted to precisely introduce this notion of effects.

2.3 Effects

The notion of effect lives at the core of F*. In F*, every computation is associated with type-level information about the nature and scope of its side effects. Each effect models a certain kind of side effects: stateful computations, divergence, exceptions, etc. Each effect comes with parameters to specify the scope of side effects. Thus every F* code fragment *lives* in a specific effect.

Section 2.3.1 presents commonly useful effects, and gives an intuition about their nature. The last two sections present more exotic effects. The first section (Section 2.3.2) presents **Tac**, an effect dedicated to F* metaprogramming. The second section (Section 2.3.3) is dedicated to effects

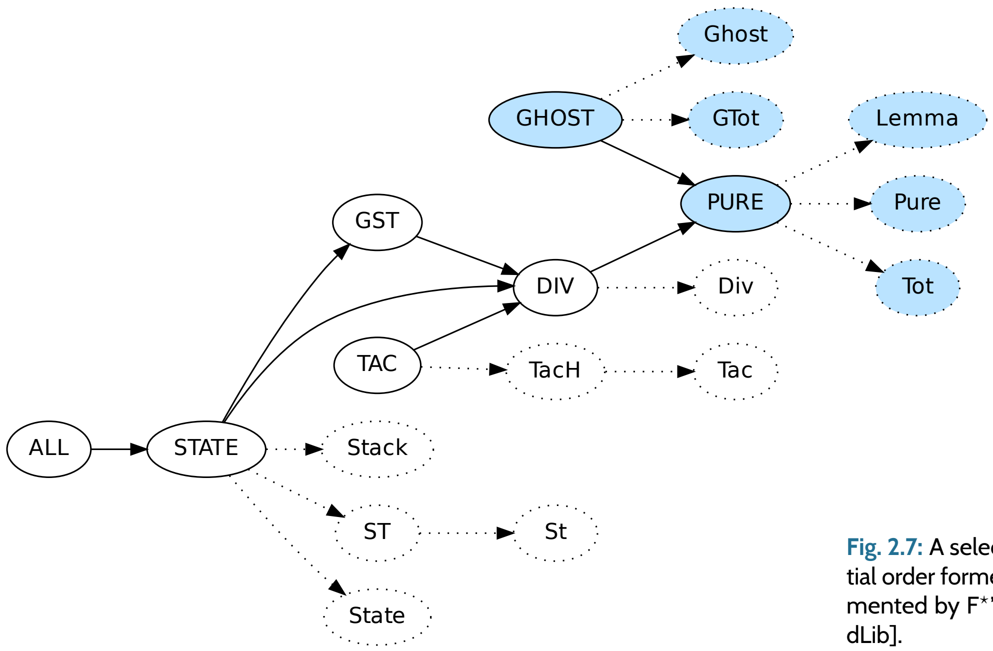

Fig. 2.7: A selected slice of the F* partial order formed by the effects implemented by F*'s standard library [FStdLib].

that implement shallow or deep embeddings. Such a thrilling effect is Low*. It makes possible low-level and C-like programming right within F*'s syntax. Under certain restrictions, the functions living in Low*'s effects can be extracted as raw C code, free of any runtime.

2.3.1 The Bestiary of F* Predefined Effects

Figure 2.7 presents the various effects available out of the box in F*. The effects with a blue background represent total computations; other ones potentially represent divergent computations. The effects represented in the figure with a dotted outline are simply reformulations of another effect. For instance, `Tot` and `Lemma` are both effect abbreviations for `PURE`. An arrow from an effect `E` to `F` means that a computation in `E` can be lifted to a computation in `F`. For instance, by transitivity, a total computation of type $x{:}\tau \rightarrow$ `Tot` τ can be lifted as a stateful computation of type $x{:}\tau \rightarrow$ `STATE` τ ..., while the other way around is not possible. The effects form a structure equipped with a partial order, allowing F* to lift computations from an effect to another in a completely automated fashion.

The `PURE` effect models pure and total computations. It is a Dijkstra monad indexed by weakest-preconditions. Figure 2.8 illustrates weakest-preconditions on pure computations. $\mathtt{div}_{\mathtt{pure}}$ and $\mathtt{div}_{\mathtt{tot}}$ divide the literal 4 by an input `n`; yielding the obligation $\mathtt{n} \neq \mathtt{0}$. The only difference between these two functions is their type annotations. The signature of $\mathtt{div}_{\mathtt{tot}}$ is straightforward: given a non-zero $\mathbb{Z}$, we get an $\mathbb{Z}$. The $\mathtt{div}_{\mathtt{tot}}$ signature is more convoluted: it takes any $\mathtt{n}{:}\mathbb{Z}$, and returns a `PURE` computation of type $\mathbb{Z}$ annotated with the weakest-precondition presented by Equation 2.5. It means that, for any post-condition `p`, if the pre-condition $\mathtt{n}{\neq}\mathtt{0} \wedge \mathtt{p}\ (4\ /\ \mathtt{n})$

```
let div_pure (n: ℤ)
  : PURE ℤ (λp → n ≠ 0
              ∧ p (4 / n))
  = 4 / n
let div_tot (n: ℤ{n≠0})
  : Tot ℤ = 4 / n
```

Fig. 2.8: Example of the same computation defined as `PURE` and then as `Tot`.

holds on `n` the input of $\texttt{div}_{\texttt{pure}}$, then $\texttt{div}_{\texttt{pure}}$ `n` admits `p` as post-condition.

$$\lambda(\texttt{p}:\ \mathbb{Z} \rightarrow \textbf{Type}_0) \rightarrow \texttt{n} \neq \texttt{0} \wedge \texttt{p (4 / n)} \tag{2.5}$$

As mentioned above, effect **`Tot`** is an abbreviation for **`PURE`**. Effect **`PURE`** exposes a weakest-precondition interface to specify pure computations, which is a bit over complicated. The interface of **`Tot`** is sufficient. As it has no side effects whatsoever, the outcome of a pure function only depends on its formal arguments. Consequently any pre- and post-condition one could express on a pure function can be encoded as refinement type, i.e.:

$$\texttt{f: x:}\tau\texttt{\{pre-condition\}} \rightarrow \texttt{r:}\beta\texttt{\{post-condition\}}$$

This observation leads to **`Tot`**'s own definition, that gives a very strong not-weakest-precondition transformer to the effect **`PURE`**, as presented in Equation 2.6. The keyword **`effect`** introduces a new effect abbreviation.

$$\textbf{effect Tot}\ \texttt{(a:}\textbf{Type}\texttt{)} = \textbf{PURE}\ \texttt{a}\ (\lambda \texttt{p} \rightarrow \forall\ \texttt{(r:a). p r)} \tag{2.6}$$

In opposition, the effect **`STATE`** presented in Figure 2.7 models stateful computation. Due to the implicit state, refinement types alone are not sufficient to express state-sensitive properties about computations in **`STATE`**.

The last effect we will present here is **`GHOST`**. It is an exact clone of **`PURE`**. The difference is their position on the partial order of effects: a **`GHOST`** computation cannot be lifted to any other effect. A computation g in **`GHOST`** is trapped in **`GHOST`**, and is marked by F* are computationally irrelevant: g will be erased at extraction. Non-constructive operations are allowed in **`GHOST`**: for instance, given a proof that $\exists$ `x. p x`, a **`GHOST`** computation is allowed to witness such a `x`.

Primitives A primitive effect is exactly a weakest-precondition monad: it provides no model of computation. An example of such an effect is **`STATE`**: as Section 2.3.3 will present, **`STATE`** is only a specification of stateful computation. The computation model of **`STATE`** is given by F* extraction to either OCaml or C.

Dijkstra Monad for Free (DM4Free) DM4Free is a way of generating both a weakest-precondition specification monad and a computational monad, bundled as an effect. DM is the input language for DM4Free, and is embedded in F*. It is a simply-typed lambda calculus. Thus the expressiveness of the representation of effects defined via DM4Free has some limitations. Defining effects in this way is now deprecated in F* in favor of *layered effects*.

Layered effects A layered effect is a full Dijkstra monad in the sense that it's a monad-like structure indexed by a specification monad. The representation of a layered effect might be an arbitrary arrow type $\ldots \rightarrow$ **`E`** $\ldots$, with effect **`E`** arbitrary. This allows for very expressive and flexible abstractions. Chapter 5 will illustrate this flexibility.

Reification and reflection Dijkstra monads for free and layered effects bring to F⋆ effects that can be viewed as computational monads indexed by specification monads. Consider **E**: $\mathsf{a}_0 : \tau_0 \to \cdots \to \mathsf{a}_n : \tau_n \to$ **Effect** an effect and `repr` $\mathsf{a}_0 \cdots \mathsf{a}_n$: **Type** its representation. The act of transforming `f`: **E** $\mathsf{a}_0 \cdots \mathsf{a}_n$ into `f'`: `repr` $\mathsf{a}_0 \cdots \mathsf{a}_n$ is called reification[8]. The opposite transformation is called reflection. An effect can be marked as reifiable and/or reflectable. Chapter 5 makes use of reification and reflection.

[8]In the context of monadic programming, such a transformation is very common. The Haskell `ST` monad is *reified* by invoking `runST` [HaskST] for instance.

2.3.2 Tactics: Manual Proving and Meta-Programming

This subsection briefly presents the **Tac** effect that hosts effectful computations that have a special role. We have seen how to prove properties about programs with the help of the SMT solver. In proof assistants like Coq, proofs are carried out by proof scripts. Such scripts are effectful computations dealing with a proof goal. A proof goal is a set of subgoals, each constituted of a set of hypotheses and of a formula to be proven.

2.3.2.1 Proving theorems in Coq with tactics

Consider the statement $\mathsf{p} \implies (\mathsf{p} \lor \mathsf{q})$, given two propositions p and q. The corresponding proof goal is presented by ❶ in Figure 2.9: the hypotheses are that p and q are propositions, and the goal is $\mathsf{p} \implies \mathsf{p} \lor \mathsf{q}$. Coq provides a number of operations that, as a side effect, manipulate the current proof goal in a sound way. Those operations are called tactics. The tactic `intro` pulls a hypothesis from an implication. Running this tactic on the proof goal ❶ given in Figure 2.9 computes no value but transforms the proof goal from ❶ to ❷ as side effect. The tactic `left` eliminates the disjunction presented in ❷ and gets us to ❸. The tactic `exact h` finishes the proof: the hypothesis h solves exactly ❸'s goal p.

Tactics are very useful, since they allow the user to incrementally build proof terms. But ultimately, Coq tactics are just producing proof terms in the calculus of (inductive) constructions [CH86]. In this sense, tactics are meta-programs: their role is to generate terms. For instance, the Coq tactics we employed on $\mathsf{p} \implies (\mathsf{p} \lor \mathsf{q})$ build the proof term $\lambda(\mathsf{x}:\mathsf{p})$ `=>` `or_introl x`, where `or_introl` is a constructor for the inductive type `or`. Given x a proof that p holds, we construct the term `or_introl x` of type $\mathsf{p} \lor \mathsf{q}$ (a.k.a. `or p q`), that is a proof of $\mathsf{p} \lor \mathsf{q}$.

While Coq programs are written in a language called Gallina, meta-programs are written in separate languages, hosted in Coq.

```
   p, q : prop
❶ ---------
   p ⟹ p ∨ q

   p, q : prop        p, q : prop
   h : p              h : p
❷ ---------        ❸ ---------
   p ∨ q              p
```

Fig. 2.9: Proof goal examples.

2.3.2.2 Tactics in F⋆

F⋆ also provides facilities to build proof terms from tactics. F⋆ tactics are regular computations, living in the effect **TAC**. This effect is defined by the F⋆ standard library [FStdLib]. The representation of a computation `f`: **TAC** τ `wp`, with `wp` a weakest-precondition, is of shape `proofstate` $\to$ `result a`, with `result a` either holding a value τ and a `proofstate`, or an error and a `proofstate`. **TAC** computations are stateful, potentially divergent, and potentially failing.

TAC is indexed by weakest-precondition; in the same spirit as PURE and Tot, the abbreviation Tac τ denotes tactics that compute values of type τ, without more specification. Another type abbreviation, TacH, allows for Hoare-style specifications for meta-programs, allowing for verified meta-programming.

2.3.2.3 Example

To illustrate how proving with tactics in F* feels, reconsider the $\texttt{mem}_0$ and `mem` functions presented in Equations 2.2 and 2.3. Below, we present Lemma 2.7 which is the manual and tactic-based alternative to the previous lemma 2.4. The aim is to prove that the functions $\texttt{mem}_0$ and `mem` are point-wise equal.

```
let rec mem_eq_tac (#a: eqtype) (l: list a) (x: a)
: Lemma (ensures (mem₀ l x == mem l x)) (decreases l)
= assert (mem₀ l x == mem l x) by (
    ❶ destruct (quote l);
    ❷ repeat' (λ_ →
        let hyps = intros () in
        ❸ guard (Cons? hyps); rewrite (last hyps);
        ❹ norm [zeta;delta_only[`%mem₀;`%mem]]; norm[iota];
        ❺ l_to_r [quote mem_eq_tac];
        ❻ trefl ()
      )
  )
```
(2.7)

Figure 2.10 presents the different states our proof `mem_eq_tac` goes through. To begin with, at ❶, the goal consists in the hypotheses that the three arguments `a`, `l` and `x` exist. Then we destruct the list `l`. Function `quote` reflects F* terms as syntactic trees (of type `term`): its type is `#a:Type → a → term`. Destructing `l` leads to the proof state ❷, composed of two subgoals: one if the list is empty, the other if it is not the case. `repeat'` takes a computation of type `unit → Tac unit` and repeats it until nothing remains to be proven.

The first repetition focuses on the goal where `l` is non-empty. The first sub-goal of ❷ presents is an arrow type: given some `hd` and some `tl`, we shall prove $\texttt{mem}_0$ `l x == mem l x`. To use `hd` and `tl` as hypotheses, we introduce them: `intros` introduces as many names as possible. Using the last hypothesis (`last hyps` in the code, and `h` in the Figure 2.10), we rewrite our goal that becomes (❹) $\texttt{mem}_0$ `(Cons hd tl) == mem (Cons hd tl)`. Unrolling $\texttt{mem}_0$ and `mem`, we get ❺. `norm` normalizes a term given a list of reductions: after δ-reducing $\texttt{mem}_0$ and `mem`, we ι-reduce to simplify superfluous `match`es. `l_to_r` rewrites every sub-term of the goal from left to right using specified lemmas. Here we recursively use our lemma `mem_eq_tac`, so that it is applied on `tl` and `x`. In consequence, $\texttt{mem}_0$ `tl x` is rewritten into `mem tl x`. ❻ presents a goal which is true by reflexivity of equality, hence we apply the tactic `trefl`.

The second repetition of `repeat'` takes care of the case where `l` is empty. After introducing the hypothesis `l==Nil`, we rewrite the proof state

❶
```
a:eqtype   l:list a   x:a
-------------------------
mem0 l x == mem l x
```

❷
```
a:eqtype   l:list a   x:a
-------------------------
hd:a → tl:list a →
l==Cons hd tl →
mem0 l x == mem l x

a:eqtype   l:list a   x:a
-------------------------
l==Nil → mem0 l x == mem l x
```

❸
```
a l x:⋯   hd:a   tl:list a
h: (l == Cons hd tl)
-------------------------
mem0 l x == mem l x
```
+ Same second goal as ❷

❹
```
a l x:⋯   hd:a   tl:list a
h: (l == Cons hd tl)
-------------------------
mem0 (Cons hd tl) x ==
mem (Cons hd tl) x
```
+ Same second goal as ❷

❺
```
a l x:⋯   hd:a   tl:list a
h: (l == Cons hd tl)
-------------------------
(hd=x || mem0 tl x) ==
(hd=x || mem tl x)
```
+ Same second goal as ❷

❻
```
a l x:⋯   hd:a   tl:list a
h: (l == Cons hd tl)
-------------------------
((hd = x || mem tl x) ==
(hd = x || mem tl x))
```
+ Same second goal as ❷

❸'
```
a l x:⋯   h:l==Nil
-------------------------
mem0 l x == mem l x
```

❹'
```
a l x:⋯   h:l==Nil
-------------------------
mem0 Nil x == mem Nil x
```

❺'
```
a l x:⋯   h:l==Nil
-------------------------
false == false
```

Fig. 2.10: Proof goals for the program `mem_eq_tac`.

```
let fact_c (n: U32.t {v n ≠ 0 ∧ v n ≤ max})
  : Stack U32.t (λ_ → ⊤)
          (λh0 r h1 → ❶ v r == fact (v n))
  = push_frame ();          // Pushes a new stack frame, solely for specification
    let r = Buffer.alloca 1ul 1ul in
    let h0 = get () in      // Gets a (computationally irrelevant) reflection of the current memory
    let inv h1 m =          // inv is the invariant our for loop below should respect
      live h1 r             // it requires the liveness of the buffer r
    ∧ modifies (loc_buffer r) h0 h1          // appart from r, nothing changes in the memory
    ∧ v (Buffer.get h1 r 0) == fact m in    // r should always be exactly fact m (m is the loop index)
    for 1ul n inv (
      λi → let r0 = !*r in
           r *= (r0 × (i + 1ul))
    );
    let r = !*r in
    pop_frame ();           // Pops the frame we pushed previously
    r
```

Fig. 2.11: Low-level verified implementation of the factorial function in F*.

❸' to rewrite l into Nil and get ❹'. mem Nil x and mem_0 Nil x both unfolds to false, as ❺' shows. The step involving l_to_r is not useful, and does nothing here. Again, the goal ❺' is trivial by reflexivity.

F* tactics are just plain computations that live in a specific effect, TAC. We saw the function quote that quotes F* terms: the standard library provides a great number of primitives to interact with F*. The meta-programming facilities combined with refinement and dependent types allow for interesting use-cases. Section 5.7 makes use of meta-programming more deeply.

2.3.3 Effects Implementing Domain-Specific Languages

Last but not least, effects can implement domain-specific languages. Such domain-specific languages then benefit from all F* capabilities in terms of verification. Low* [Pro+17] is an F* library that models the C memory model and provides effects allowing C-like low-level programming as a shallow embedding in F*. Let us dive directly into an example with Figure 2.11 which presents an efficient low-level implementation of our function fact. In supplement to Low*, KreMLin [Pro+17] is a tool that extracts an F* program as C code. KreMLin won't extract any F* program; it requires the target program to respect certain restrictions implemented in part by Low*'s effects. In addition, certain features for which there exists no clear C counterpart, i.e. higher-order functions, are forbidden by KreMLin. Figure 2.12 presents the C code generated by KreMLin for the Low* factorial function of Figure 2.11.

Low* programs can however rely on the full F* feature set (i.e. higher-order functions) when it comes to specifications. For example, in Figure 2.11, the specification ❶ of fact_c is given by reflecting our previous functional implementation fact from Equation 2.1.

Low* has been very effective: for example, it enabled the formal verified implementation of HACL* [Bha+17], a cryptographic library deployed for

```c
uint32_t fact_c(uint32_t n)
{
  uint32_t r = (uint32_t)1U;
  for (uint32_t i = (uint32_t)1U; i < n; i++){
    uint32_t r0 = *&r;
    *&r = r0 * (i + (uint32_t)1U);
  }
  uint32_t r1 = *&r;
  return r1;
}
```

Fig. 2.12: C code generated from the extraction of the F⋆ factorial function of figure 2.11.

example in Firefox or Wireguard VPN. KreMLin can also directly compile Low⋆ to WebAssmbly [Pro+19].

In a similar way, the Vale project [PEVale21] includes an embedding of x64 assembly [Fro+19] in F⋆. Another interesting F⋆ DSL is EverParse [Ram+19], which is a parser generator for binary data formats. Finally, Steel [Swa+20] is an ongoing effort to embed a concurrent separation logic framework in F*.

All these examples illustrate the wide range of scenarios F⋆ can be used for. Thanks to its effects, F⋆ is very versatile and can be used for a great number of situations.

2.4 Conclusion

This section has presented a selection of features F⋆ offers. Through refinement (Section 2.1.1) and dependent types (Section 2.1.2), F⋆ allows to state and verify property about programs in a flexible and powerful way. Its focus on effects and on Dijkstra monads (Section 2.2) allows verification for computations which have all kinds of side effects. Most of the time, F⋆ verification conditions are automatically discharged by SMT solver techniques (Section 2.1). When automation is too weak, proofs can be written, entirely or (more commonly) partially, in a more classical proof assistant style with proof scripts (Section 2.3.2). Programs written and proven correct in F⋆ can be extracted as OCaml programs. A number of F⋆ shallow embeddings (Section 2.3.3) allows to write and prove low-level programs and to extract them as, i.e., C, WebAssembly, or x64 assembly.

Those different features make F⋆ very interesting and promising for writing real-world verified software. Such capabilities have been demonstrated with HACL* for instance. The picture is not all bright however; certain patterns, such as stateful code or recursion, require from the programmer some rather boring annotation. F⋆ could use some help from static analysis approaches. Chapter 3 presents a static analyzer implemented and verified in F⋆. Then, the Chapter 4 establishes a hybridization of such verified F⋆ analyser with F⋆ effects, with the aim of lightening the annotation effort of F⋆ programmers.

CHAPTER 3

An Abstract Intepreter Verified With F*

In Chapter 2, we presented the dependently typed language F*. A language equipped with a very powerful and precise type system is one of the ways to achieve formal verification of programs. In the case of such a programming language, a program implementation directly embeds its specifications. Each code fragment is typed, and thus has a specification. In such settings, programs are specified and implemented at the same time.

By contrast, most other approaches to formal verification dissociate implementation from verification. Such formal verification tools build a mathematical model from an existing implementation of a program. This mathematical model –faithful to the implementation semantics– is then used to *certify* the implementation as respecting certain given properties. The workflow thus consists first in writing a program, then in building a faithful model of its semantics, to finally verify whether a property holds.

This chapter focuses on *abstract interpretation*. Most of the tools that follow the methodology of abstract interpretation do not formally establish a relation between their algorithmic theory and implementations. Several abstract interpreters have however been proven correct. The most notable one is Verasco [Jou+], a static analyser of C programs that has been entirely written, specified and proven in the proof assistant Coq. However, understanding the implementation and proof of Verasco requires an expertise with Coq and proof assistants.

Proofs in Coq are achieved thanks to an extensive use of proof scripts, that are very difficult for non expert to read. By contrast with a handwritten proof, a Coq proof can be very verbose, and does often not convey a good intuition for the idea behind a proof. Thus, writing and proving sound a static analyzer is a complex and time-consuming task: for example, Verasco requires about 17k lines [Jou+] of manual Coq proofs. Such an effort, however, yields the strongest guarantees and provides complete trust in the static analyser.

This chapter showcases the F* implementation of a sound static analyser, by presenting about 95% of its 527 lines of code. It is an abstract interpreter equipped with the numerical abstract domain of intervals, forward and backward analyses of expressions, widening, and syntax-directed loop iteration. The implementation we present is the first abstract interpreter verified with SMT techniques. We gain an order of magnitude in the number of proof lines in comparison with similar works with Coq implementations.

Contents

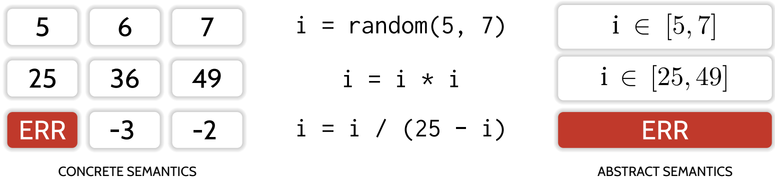

Fig. 3.2: Concrete and abstract interpretation of a simple program.

3.1 Some Intuition About Abstract Interpretation

Abstract interpretation is a theory of sound approximation of program semantics. A standard program interpreter runs a program in a concrete world: the possibly wide domain of inputs and possible non-determinism may yield an enormous number of possible execution trajectories. The white lines of Figure 3.1 represent such many concrete trajectories for a variable x. By contrast, an abstract interpreter runs that same program in an abstract world: instead of mapping variables to precise concrete values, variables are mapped to (sound) sets of possible values. The shape of Figure 3.1 represents such an approximation over time for a variable x. Figure 3.2 presents the concrete and abstract interpretation for a 3 instructions program which consists in arithmetic operations on a random number between 5 and 7. There exists three different paths for its concrete interpretations. Instead of interpreting the program with concrete values, the abstract interpreter approximates the program with abstract values. In this example, the abstract values are intervals of numbers: the interval $[5, 7]$ describes exactly the set of concrete values $\{5, 6, 7\}$, and the interval $[25, 49]$ is an over approximation of the concrete values 25, 36 and 49. Here, it is possible to run a concrete interpretation for every possible path. The concrete interpretation would be sufficient here to spot the division by zero.

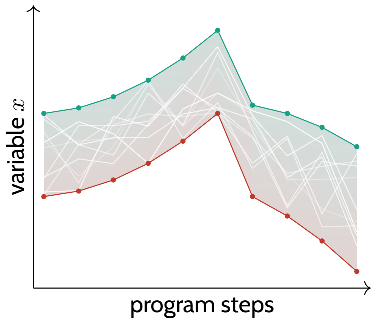

Fig. 3.1: An abstract interpreter catches the many possible concrete semantics (the white lines) by approximating them (the colored shape).

Now, consider the program in Figure 3.4. The integer `N` being an arbitrarily large integer, the number of possible concrete interpretations is possibly very big. For `N` sufficiently big, running all the possible concrete interpretations is impossible in a reasonable amount of time. On the abstract interpreter side however, this is not a problem: the analysis is still performed in 3 steps. Indeed, a random number between 5 and `N` is approximated by $[5, \infty[$, which leads to the division by zero being spotted. This is the power of abstract interpretation: it provides an approximate but sound interpretation of a program in finite time.

Abstract interpretation is a sound theory, meaning that if the analysis finds no runtime error, then the program won't fail at runtime. The other way around is not true: an abstract interpreter approximates the semantics of a program, and thus misses some of its subtleties. Its approximate analysis might yield a *false alarm*, warning about an issue that will not occur in practice. Figure 3.3 illustrates such a false alarm: $[-1, 1]$ over approximates the concrete values 1 and -1, and the abstract interpreter warns about a (possible) division by zero.

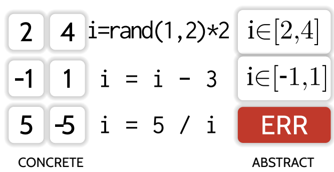

Fig. 3.3: Example of a false alarm.

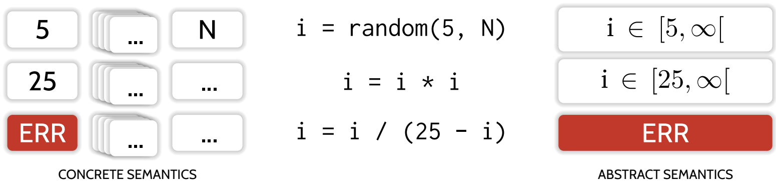

Fig. 3.4: Concrete and abstract interpretation of a simple program with more possible execution paths. N is an arbitrarily large integer.

3.2 IMP: a Small Imperative Language

To present our abstract interpreter, we first show the language on which it operates: IMP. It is a simple imperative language, equipped with memories represented as functions from variable names `varname` to signed integers, $\texttt{int}_\texttt{m}$. This chapter is also an opportunity to examplify some aspects of F* presented in Chapter 2. IMP's F* definition looks like OCaml; the main difference is the explicit type signatures for constructors in algebraic data types. IMP has numeric expressions, encoded by the type `expr`, and statements `stmt`. Booleans are represented numerically: 0 represents `false`, and any other value stands for true. The enumeration `binop` equips IMP with various binary operations. The constructor **`Unknown`** encodes an arbitrary number. Statements in IMP are the assignment, the non-deterministic choice[1], the sequence and the loop.

```
type varname = | VA | VB | VC | VD
type mem τ = varname → τ
type binop = | Plus | Minus | Mult | Eq | Lt | And | Or
type expr = | Const: int_m → expr    | Var: varname → expr
            | BinOp: binop → expr → expr → expr | Unknown
type stmt = | Assign: varname → expr → stmt
            | Assume: expr → stmt | Loop: stmt → stmt
            | Seq:    stmt → stmt → stmt
            | Choice: stmt → stmt → stmt
```

[1]The conditional **`if`** ⟦c⟧ **`then`** ⟦a⟧ **`else`** ⟦b⟧ is thus represented as a non-deterministic choice between `Seq (Assume c) a` and `Seq (Assume c') b`, with `c'` the negation of `c`.

The type $\texttt{int}_\texttt{m}$ is a refinement of the built-in F* type $\mathbb{Z}$: while every integer lives in the type $\mathbb{Z}$, only those that respect certain bounds live in $\texttt{int}_\texttt{m}$. Numerical operations ($+$, $-$ and $\times$) on machine integers wrap on overflow, i.e. adding one to the maximal machine integer results in the minimum machine integer. We do not give the detail of their implementation.

3.3 Operational Semantics

This section defines an operational semantics for IMP. We choose to formulate our semantics in terms of sets. Sets are encoded as maps from values to propositions `prop`. Those are logical statements and shouldn't be confused with booleans. Below, $\subseteq$ quantifies over every *inhabitant* of a type: stating whether such a statement is true or false is clearly not computable.

Arbitrarily complex properties can be expressed as propositions of type `prop`.

In the listing below, notice the Greek letters: we use them throughout the manuscript. They denote implicit type arguments: for instance, below, ∈ works for any set `set τ`, with any type τ. F* provides the propositional operators ∧, ∨ and ==, in addition to boolean ones (&&, || and =). We use them below to define the union, intersection and differences of sets.

```
type set τ = τ → prop
let (∈) (x: τ) (s: set τ) = s x
let (∩) s0 s1 = λx→x∈s0 ∧ x∈s1
let (\) s0 s1 = λv→s0 v ∧ ~(s1 v)
let (∪) (s0 s1: set τ): set τ = λx → x ∈ s0 ∨ x ∈ s1
let (⊆) (s0 s1: set τ): prop = ∀ (x: τ). x ∈ s0 ⟹ x ∈ s1
let set_inverse (s: set int_m): set int_m = λ(i: int_m) → s (-i)
```

To be able to work conveniently with binary operations on integers in our semantics, we define `lift_binop`, that lifts them as set operations. For example, the set `lift_binop (+) a b` (a and b being two sets of integers) corresponds to $\{va + vb \mid va \in \mathtt{a} \land vb \in \mathtt{b}\}$.

```
let lift_binop (op: τ → τ → τ) (a b: set τ): set τ
  = λr → ∃ (va:τ). ∃ (vb:τ). va ∈ a ∧ vb ∈ b ∧ r == op va vb
unfold let lift op = lift_binop (concrete_binop op)
```

The binary operations we consider are enumerated by `binop`. The function `concrete_binop` associates these syntactic operations to integer operations. For convenience, `lift` maps a `binop` to a set operation, using `lift_binop`. This function is directly inlined by F* when used because of the keyword **unfold**; intuitively `lift` behaves as a macro.

```
unfold let concrete_binop (op: binop): int_m → int_m → int_m
  = match op with | Plus → n_add | Lt → lt_m | ... | Or → ori_m
```

The operational semantics for expressions is given as a map from memories and expressions to sets of integers. Notice the use of both the syntax **val** and **let** for the function $\mathtt{osem}_{\mathtt{expr}}$. The **val** syntax gives $\mathtt{osem}_{\mathtt{expr}}$ the type `mem→expr→set int_m`, while the **let** declaration gives its definition. The semantics itself is uncomplicated: **Unknown** returns the set of every $\mathtt{int}_\mathtt{m}$, a constant or a **Var** returns a singleton set. For binary operations, we lift them as set operations, and make use of recursion.

```
val osem_expr: mem → expr → set int_m
let rec osem_expr m e = λ(i: int_m)
  → match e with | Const x → i==x | Var v → i==m v | Unknown → ⊤
  | BinOp op x y → lift op (osem_expr m x) (osem_expr m y) i
```

The operational semantics for statements maps a statement and an initial memory to a set of admissible final memories. Given a statement `s`, an initial memory $\mathtt{m_i}$ and a final one $\mathtt{m_f}$, $\mathtt{osem}_{\mathtt{stmt}}\ \mathtt{s}\ \mathtt{m_i}\ \mathtt{m_f}$ (defined below) is a proposition stating whether the transition is possible.

```
val osem_stmt (s: stmt): mem → set mem
let rec osem_stmt (s: stmt) (m_i m_f: mem)
  = match s with
```

```
| Assign v e → ∀w. if v = w then mf v ∈ osem_expr mi e
                            else mf w == mi w
| Seq a b  → ∃ (m1: mem). m1 ∈ osem_stmt a mi
                        ∧ mf ∈ osem_stmt b m1
| Choice a b → mf ∈ (osem_stmt a mi ∪ osem_stmt b mi)
| Assume e → mi == mf
           ∧ (∃ (x: int_m). x ≠ 0 ∧ x ∈ osem_expr mi e)
| Loop a → closure (osem_stmt a) mi mf
```

The simplest operation is the assignment of a variable v to an expression e: the transition is allowed if every variable but v in $\mathtt{m_i}$ and $\mathtt{m_f}$ is equal and if the final value of v matches with the semantics of e. Assuming that an expression is true amounts to require the initial memory to be such that at least a non-zero integer (that is, the encoding of `true`) belongs to $\mathtt{osem_{expr}}$ $\mathtt{m_i}$ e. As Figure 3.5 illustrates, the statement **Seq** a b starting from the initial memory $\mathtt{m_i}$ admits $\mathtt{m_f}$ as a final memory when there exists (i) a transition from $\mathtt{m_i}$ to an intermediate memory $\mathtt{m_1}$ with statement a and (ii) a transition from $\mathtt{m_1}$ to $\mathtt{m_f}$ with statement b. The operational semantics for a loop is defined as the reflexive transitive closure of the semantics of its body. The `closure` function computes such a closure, and is provided by F* standard library.

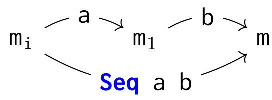

Fig. 3.5: Illustration of the operational semantics for the statement **Seq** a b.

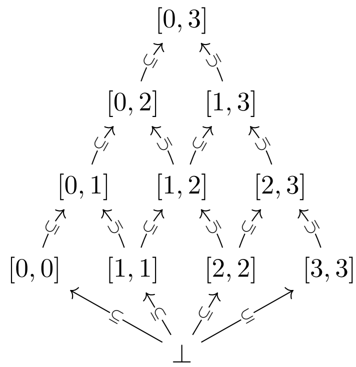

Fig. 3.6: The lattice of intervals of integers between 0 and 3.

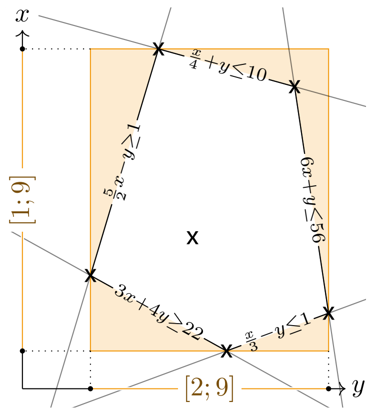

Fig. 3.7: Abstractions of the variables x and y; the possible concrete trajectories for x are $\{2,4,6,8,9\}$, and the possible ones for y are $\{1,2,3,8,9\}$. The orange rectangle illustrates the interval abstraction $[2;9]$ for x and the one $[1;9]$ for y. The black polygon represents an abstraction for (x,y) in the domain of polyedra.

3.4 Abstract Domains

The core of an abstract interpreter is its abstract domain. Section 3.1 presents a few examples relying on intervals as an abstract domain for integers. Interval $[1;3]$ describes the set of values $\{1;2;3\}$ and is thus a sound over-approximation of, e.g. the value 2. An abstract value is to be seen as a property: $[1;3]$ meaning “between 1 and 3”. Each abstract domain has its own expressivity in terms of invariants: for instance, the interval domain can only represent a limited class of properties, e.g., the range of variables.

Abstract interpretation of programs computes abstract values instead of concrete ones. Abstract domains are lattices partially ordered by a relation $\sqsubseteq^{\#}$ that models properties entailment. For instance, in the lattice presented in 3.6, $[0;3]$ is greater than $[1;3]$: if “between 1 and 3” holds on some x, then “between 0 and 3” holds on x as well. Consider the lattice $(\mathcal{P}(\mathbb{Z}),\subseteq)$ of concrete properties on numerical values and another one $(D^{\#},\sqsubseteq^{\#})$, the abstract domain, for some $D^{\#}$ set. Suppose $D^{\#}$ is an abstract domain that approximates $\mathcal{P}(\mathbb{Z})$, the concrete domain of integers. Then, an abstraction function α maps concrete properties $x \in \mathcal{P}(\mathbb{Z})$ to $\alpha(x) \in D^{\#}$, its best abstract approximations. The concretization function γ does the opposite. A concrete interpreter associates a computation F to a partial function $f : \mathbb{Z} \to \mathbb{Z}$, whereas an abstract interpreter associates F to a total function $f^{\#} : D^{\#} \to D^{\#}$.

There exists a lot of different domains with different expressivities. The domain of intervals is non relational: it cannot express relationship between variables. Figure 3.7 illustrates the gap of expressivity between non-relational domains and relational ones.

Parametricity Our abstract interpreter is parametrized over relational domains. We instantiate it later with a weakly-relational[2] [Cou+05] memory. This section defines lattices and abstract domains. Such structures are a natural fit for typeclasses [Mar+19], which allow for ad hoc polymorphism. In our case, it means that we can have one abstraction for lattices for instance, and then instantiate this abstraction with implementations for, say, sets of integers, then intervals, etc. Typeclasses can be seen as record types with dedicated dependency inference. Below, we define the typeclass `lattice`: defining an instance for a given type equips this type with a lattice structure.

[2] A non relational domain augmented with a relational but weak property is called weakly-relational. Later, we consider abstract memories, mapping variables to intervals. We augment this mapping with a $\perp$ element, encoding unreachability, this property being relational.

Refinement types Below, the syntax `x:τ{p x}` denotes the type whose inhabitants both belong to τ and satisfy the predicate `p`. For example, the only inhabitant of the type `bot:ℕ{∀(n:ℕ). bot≤n}` is `0`, the smallest natural number. To typecheck `x:τ`, F* collects the *proof obligations* implied by "`x` has the type τ", and tries to discharge them with the help of the SMT solver. If the SMT solver is able to deal with the proof obligations, then `x:τ` typechecks. In the case of "`0` is of type `bot:ℕ{∀(n:ℕ). bot≤n}`", the proof obligation is `∀(n:ℕ). 0≤n`.

Below, most of the types of the fields from the record type `lattice` are refined. Typechecking `i` against the type `lattice τ` yields a proof obligation asking (among other things) for `i.join` to go up in the lattice and for `bottom` to be a lower bound. Thus, if "`i` has type `lattice τ`" typechecks, it means that there exists a proof of the properties on `i` written as refinements in the definition of `lattice`. We found convenient to let `bottom` represent unreachable states. Note `lattice` is under-specified, i.e. it doesn't require `join` to be provably a least upper bound, since such a property plays no role in our proof of soundness. This choice follows Blazy and al. [BLP16].

```
class lattice τ = { corder: order τ
  ; join: x:τ → y:τ → r:τ {corder x r ∧ corder y r}
  ; meet: x:τ → y:τ → r:τ {corder r x ∧ corder r y}
  ; bottom: bot:τ{∀x. corder bot x}; top: top:τ{∀x. corder x top}}
```

For our purpose, we need to define what an abstract domain is. In our setting, we consider concrete domains with powerset structure. The typeclass `adom` encodes them: it is parametrized by a type τ of abstract values. For instance, consider `itv` the type for intervals: `adom itv` would be the type inhabited by correct abstract domains for intervals.

Implementing an abstract domain amounts to implementing the following fields: (i) `c`, that represents the type to which abstract values τ concretize; (ii) $\mathtt{adom}_{\mathtt{lat}}$, a lattice for τ; (iii) `widen`, a widening operator that ensures convergence of fixpoint iterations; (iv) γ, a monotonic concretization function from τ to `set c`; (v) `order_measure`, a measure ensuring the abstract domain doesn't admit infinite increasing chains, so that termination is provable for fixpoint iterations; (vi) $\mathtt{meet}_{\mathtt{law}}$, that requires `meet` to be a correct approximation of set intersection; (vii) $\mathtt{top}_{\mathtt{law}}$ and $\mathtt{bot}_{\mathtt{law}}$, that ensure the lattice bottom concretization matches with the empty set, and similarly for top.

```
class adom τ = { c: Type; adom_lat: lattice τ
  ; γ: (γ: (τ → set c) {∀ (x y: τ). corder x y ⟹ (γ x ⊆ γ y)})
  ; widen: x:τ → y:τ → r:τ {corder x r ∧ corder y r}
  ; order_measure: measure adom_lat.corder
  ; meet_law: x:τ → y:τ → Lemma ((γ x ∩ γ y) ⊆ γ (meet x y))
  ; bot_law: unit → Lemma (∀ (x:c). ~(x ∈ γ bottom))
  ; top_law: unit → Lemma (∀ (x:c). x ∈ γ top)}
```

Notice the refinement types: we require for instance the monotony of γ. Every single instance for adom will be checked against these specifications. No instance of adom where γ is not monotonic can exist. Given a proposition p, the **Lemma** p syntax signals a function whose outcome is computationally irrelevant, since it simply produces (), the inhabitant of type **unit**. However, as Section 2.1.3 explains, it does not produce an arbitrary **unit**: it produces an inhabitant of _:**unit** {p}, that is, the type **unit** refined with the goal p of the lemma itself.

For practicality, we define some infix operators for $\mathtt{adom}_{\mathtt{lat}}$ functions. The syntax {|...|} lets one formulate typeclass constraints: for example, (⊑) below ask F* to resolve an instance of the typeclass adom for the type τ, and name it l. Below, (⊓) instantiates the lemma $\mathtt{meet}_{\mathtt{law}}$ explicitly: $\mathtt{meet}_{\mathtt{law}}$ x y is a unit value that carries a proof in the type system.

```
let (⊑) {|l:adom τ|} = l.adom_lat.corder
let (⊔) {|l:adom τ|} (x y:τ): r:τ { corder x r ∧ corder y r
                                     ∧ (γ x ∪ γ y) ⊆ γ r } = join x y
let (⊓) {|l:adom τ|} (x y:τ): r:τ { corder r x ∧ corder r y
                                     ∧ (γ x ∩ γ y) ⊆ γ r }
  = let _ = meet_law x y in meet x y
```

Lemmas are functions that produce refined **unit** values carrying proofs. Below, given an abstract domain i, and two abstract values x and y, join_lemma i x y is a proof concerning i, x and y. Such an instantiation can be manual (i.e. below, i.$\mathtt{top}_{\mathtt{law}}$ () in top_lemma), or automatic. The automatic instantiation of a lemma is decided by the SMT solver. Below, we make use of the **SMTPat** syntax, that allows us to provide the SMT solver with a list of patterns. Whenever the SMT solver matches a pattern from the list, it instantiates the lemma in stake. The lemma join_lemma below states that the union of the concretization of two abstract values x and y is below the concretization of the abstract join of x and y. This is true because of γ monotony: we help the SMT solver a little by giving a hint with **assert**. This lemma is instantiated every time a proof goal contains x ⊑ y.

Because of a technical limitation, we cannot write SMT patterns directly in the $\mathtt{meet}_{\mathtt{law}}$, $\mathtt{bot}_{\mathtt{law}}$ and $\mathtt{top}_{\mathtt{law}}$ fields of the class adom: below we thus reformulate them.

```
let top_lemma (i: adom τ)       (let bot_lemma, meet_lemma = ...)
  : Lemma (∀ (x: i.c). x ∈ i.γ i.adom_lat.top)
```

```
        [SMTPat (i.γ i.adom_lat.top)] = i.top_law ()
let join_lemma (i: adom τ) (x y: τ)
  : Lemma ((i.γ x ∪ i.γ y) ⊆ i.γ (i.adom_lat.join x y))
        [SMTPat (i.adom_lat.join x y)]
  = let r = i.adom_lat.join x y in assert (γ x ⊆ γ r ∧ γ y ⊆ γ r)
```

3.5 An Example of Abstract Domain: Intervals

Until now, we mostly presented specificational aspects of our abstract interpreter. This section presents the abstract domain of intervals, and thus shows how proof obligations are dealt with in F*.

3.5.1 Definition of Intervals

Below, the type `itv'` is a dependent tuple: the refinement type on its right-hand side component up depends on low. If a pair ⦇x, y⦈ is of type `itv'`, we have a proof that $x \leq y$. Function `dfst` takes the first element of a dependent tuple, `dsnd` the second one.

```
type itv' = low:int_m & up:int_m {low≤up}     type itv = withbot itv'
```

The machine integers being finite, `itv'` naturally has a top element. However, `itv'` cannot represent the empty set of integers, whence `itv`, that adds an explicit bottom element using `withbot`. For convenience, `mk` makes an interval out of two numbers, and $\mathtt{itv}_{\mathtt{card}}$ computes the cardinality of an interval. We will use it later to define a measure for intervals. `inbounds x` hold when $x{:}\mathbb{Z}$ fits machine integer bounds.

```
type withbot (a: Type) = | Val: v:a → withbot a | Bot
let mk (x y: ℤ): itv = if inbounds x && inbounds y && x ≤ y
                         then Val ⦇x,y⦈ else Bot
let itv_card (i:itv):ℕ = match i with | Bot→0 | Val i→dsnd i - dfst i + 1
```

Below, $\mathtt{lat}_{\mathtt{itv}}$ is an instance of the typeclass `lattice` for intervals: intervals are ordered by inclusion, the `meet` and `join` operations consist in unwrapping `withbot`, then playing with bounds. $\mathtt{lat}_{\mathtt{itv}}$ is of type `lattice itv`: it means for instance that we have the proof that the join and meet operators respect the order $\mathtt{lat}_{\mathtt{itv}}\mathtt{.corder}$, as stated in the definition of `lattice`. Note that, here, not a single line of proofs is required: F* transparently builds up proof obligations, and asks the SMT to discharge them, which does so automatically.

```
instance lat_itv: lattice itv =
  { corder = withbot_ord #itv' (λ⦇a,b⦈ ⦇c,d⦈ → a≥c && b≤d)
  ; join = (λ(i j: itv) → match i, j with
       | Bot, k | k, Bot → k
       | Val ⦇a,b⦈, Val ⦇c,d⦈ → Val ⦇min a c, max b d⦈)
  ; meet = (λ(x y: itv) → match x, y with
```

```
  | Val ⦇a,b⦈, Val ⦇c,d⦈ → mk (max a c) (min b d)
  | _ → Bot); bottom = Bot; top = mk min_int_m max_int_m }
```

3.5.2 Fixpoint Iterations With a Widening Operator

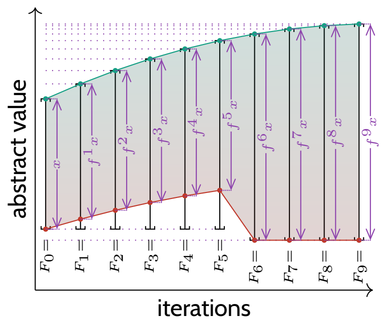

Fig. 3.8: Abstract interpretation of the iteration of abstract operator f, starting at interval x. The sequence F is defined by $F_0 = x$ and $F_{i+1} = F_i \cup f\ F_i$. Thus F is strictly increasing: it accumulates properties. Here, F^7 is a fixpoint: for any i, the approximation $f^i\ x$ is contained in F^7.

To reason about loops, loop invariants are of particular interest. An invariant is a property that holds before and after each iteration. As exposed in Section 3.1, an abstract value can be interpreted as a property. In the settings of abstract interpretation, we are hence looking for abstractions capturing loop invariants. Figure 3.8 illustrates the computation of a fixpoint. In certain abstract domains, there exists infinite increasing chains (of such iterations with respect to the lattice order): in such cases, a fixpoint iteration as presented in Figure 3.8 would never end. Even if in our case the lattice of intervals over bounded integer is finite, such iterations can be slow (i.e. proportional to the height of the lattice). To prevent non-convergent or slow fixpoint computation in abstract domains, we need to make use of *widening* [CC77].

Section 3.6 presents an F* formalization of widening operators. For the sake of simplicity however, our abstract interpreter assumes its abstract domains to be finite. In such settings, widening operators trivially converge, thus we avoid the complexity addressed in Section 3.6.

Below, `widen` implements a very classical widening operator for intervals, based on thresholds. Without a single line of proof, `widen` is shown as respecting the order `corder`.

```
let thresholds: list int_m
  = [min_int_m;-64;-32;-16;-8;-4;4;8;16;32;64;max_int_m]
let widen_bound_r (b: int_m): (r:int_m {r>b ∨ b=max_int_m}) =
  if b=max_int_m then b
  else find' (λ(u:int_m) → u>b) thresholds
let widen_bound_l (b: int_m): (r:int_m {r<b ∨ b=min_int_m}) =
  if b=min_int_m then b
  else find' (λ(u:int_m) → u<b) (rev thresholds)
let widen (i j: itv): r:itv {corder i r ∧ corder j r}
  = match i, j with | Bot, x | x, Bot → x
  | Val⦇a,b⦈,Val⦇c,d⦈ →
        Val ⦇ (if a≤c then a else widen_bound_l c)
           , (if b≥d then b else widen_bound_r d)⦈
```

Similarly, turning `itv` into an abstract domain requires no proof effort. Below $\texttt{itv}_{\texttt{adom}}$ explains that intervals concretize to machine integers ($\texttt{c} = \texttt{int}_\texttt{m}$), how it does so (with $\gamma = \texttt{itv}_\gamma$), and which lattice is associated with the abstract domain ($\texttt{adom}_\texttt{lat} = \texttt{lat}_\texttt{itv}$). As explained previously, the proof of a proposition p in F* can be encoded as an inhabitant of a refinement of **unit**, whence the "empty" lambdas: we let the SMT solver figure out the proof on its own.

```
let itv_γ: itv→set int_m
  =withbot_γ (λ(i:itv')x→dfst i≤x ∧ x≤dsnd i)
instance itv_adom: adom itv =
  { c = int_m; adom_lat = lat_itv; γ = itv_γ
```

```
; meetₗₐw = (λ_ _ → ()); botₗₐw = (λ_ → ()); topₗₐw = (λ_ → ())
; widen = widen; order_measure={f=itv_card;max=size_int_m}}
```

3.5.3 Forward Binary Operations on Intervals

This subsection takes care of defining common arithmetic and logical operators for intervals. Most of these binary operators on intervals can be written and shown correct without any proof. Our operators handle overlows of machine integers. For instance, `add_overflows` returns a boolean indicating whether the addition of two integers overflows, solely by performing machine integer operations.

Arithmetic operations

The refinement of `add_overflows` states that the returned boolean `r` should be true if and only if the addition in $\mathbb{Z}$ differs from the one in $\texttt{int}_\texttt{m}$. The correctness of $\texttt{itv}_\text{add}$ is specified as a refinement: the set of the additions between the concretized values from the input intervals is to be included in the concretization of the abstract addition. Its implementation is very simple, and its correctness is proved automatically.

```
let add_overflows (a b: int_m)
  : (r: bool {r ⟺ int_arith.n_add a b ≠ int_m_arith.n_add a b})
  = ((b<0) = (a<0)) && abs a > max_int_m - abs b
let itv_add (x y: itv): (r: itv {(γ x + γ y) ⊆ γ r})
  = match x, y with | Val ⦇a, b⦈, Val ⦇c, d⦈
                     → if add_overflows a c || add_overflows b d
                       then top else Val ⦇a + c, b + d⦈ | _→Bot
```

However in the case of interval inversion, the SMT solver sometimes misses a necessary lemma, for which we give a tactic-based proof below (as explained in Section 2.3.2). Everything within the parenthesis following the **by** keyword is a tactic. It proves that subtracting two numerical sets a and b is equivalent to adding a with the inverse of b.

Unfortunately, due to the nature of `lift_binop`, this yields existential quantifications which are difficult for the SMT solver to deal with. After normalizing our goal (with `compute ()`), and dealing with quantifiers and implications (`forall_intro`, `implies_intro` and `elim_exists`), we are left with `∃y. b (-y) ∧ r=x+y` knowing `b z ∧ r=x-z` given some `z` as an hypothesis. Eliminating `∃y` with `-z` is enough to complete the proof. This showcases the power and flexibility of F* type system: one can state arbitrarily mathematically-hard propositions (for which automation is hopeless). In such cases, one can always resort to Coq-like manual proving to handle hard proofs.

```
let set_inverse (s: set int_m): set int_m = λ(i: int_m) → s (-i)
let lemma_inv (a b: set int_m)
  : Lemma ((a-b) ⊆ (a+set_inverse b)) [SMTPat (a+set_inverse b)]
  = assert ((a-b) ⊆ (a+set_inverse b)) by (  compute ();
      let _= forall_intro () in let p0 = implies_intro () in
      let witX,p1 = elim_exists (binder_to_term p0) in
```

```
let witY,p₁ = elim_exists (binder_to_term p₁) in
let z: ℤ = unquote (binder_to_term witY) in
witness witX; witness (quote (-z)))
```

Notice the SMT pattern: the lemma $\texttt{lemma}_{\texttt{inv}}$ will be instantiated every time the SMT deals with an addition involving an inverse. Defining the subtraction $\texttt{itv}_{\texttt{sub}}$ is a breeze: it simply performs an interval addition and an interval inversion. Here, no need for a single line of proof for its correctness (expressed as a refinement).

```
let itv_inv (i: itv): (r: itv {set_inverse (γ i) ⊆ γ r})
  = match i with | Val⦇lower, upper⦈ → Val⦇-upper, -lower⦈ | _ → i
let itv_sub (x y:itv):(r:itv{(γ x - γ y) ⊆ γ r}) = itv_add x (itv_inv y)
```

Proving multiplication sound on intervals requires a lemma which is not automatically inferred:

$$\forall x \in [a,b], y \in [b,c].x \times y \in [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)]$$

In that case, decomposing that latter lemma into sublemmas $\texttt{lemma}_{\texttt{min}}$ and $\texttt{lemma}_{\texttt{mul}}$ is enough. Apart from this lemma, $\texttt{itv}_{\texttt{mul}}$ is free of any proof term.

```
let lemma_min (a b c d: ℤ) (x: ℤ{a≤x ∧ x≤b}) (y: ℤ{c≤y ∧ y≤d})
  : Lemma (x×y≥a×c ∨ x×y≥a×d ∨ x×y≥b×c ∨ x×y≥b×d) = ()
unfold let in_btw (x: ℤ) (l u: ℤ) = l≤u ∧ x≥l ∧ x≤u
let lemma_mul (a b c d x y: ℤ)
  : Lemma (requires in_btw x a b ∧ in_btw y c d)
    (ensures x×y ≥ (a×c) `min` (a×d) `min` (b×c) `min` (b×d)
           ∧ x×y ≤ (a×c) `max` (a×d) `max` (b×c) `max` (b×d))
    [SMTPat (x×y); SMTPat (a×c); SMTPat (b × d)]
  = lemma_min a b c d x y; lemma_min (-b) (-a) c d (-x) y

let mul_overflows (a b:int_m):(r:bool{r≠inbounds (int_arith.n_mul a b)})
  = a ≠ 0 && abs b > max_int_m `div_m` (abs a)
let itv_mul (x y: itv): r:itv {(γ x × γ y) ⊆ γ r}
  = match x, y with
    | Val ⦇a, b⦈, Val ⦇c, d⦈ →
        let l = (a×c) `min` (a×d) `min` (b×c) `min` (b×d) in
        let r = (a×c) `max` (a×d) `max` (b×c) `max` (b×d) in
        if mul_overflows a c || mul_overflows a d
        || mul_overflows b c || mul_overflows b d
        then top else Val ⦇l, r⦈
    | _ → Bot
```

The syntax `` `f` `` denotes the infix notation for function `f`. For instance, ``x `f` y`` is desugared into `f x y`.

Logical operations

The forward boolean operators for intervals require no proof at all. Booleans being represented by integers, `itv_as_bool` returns **TT** when an interval does not contain 0, **FF** when it is the singleton 0, **Unk** otherwise. This behavior is illustrated by Figure 3.9.

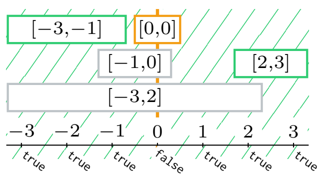

Fig. 3.9: Behavior of the function `itv_as_bool`. The interval in orange is recognized as `false` (**FF**), the green ones as `true` (**TT**). The intervals in gray abstract both integers representing `true` and the ones representing `false`: their boolean value is unknown (**Unk**).

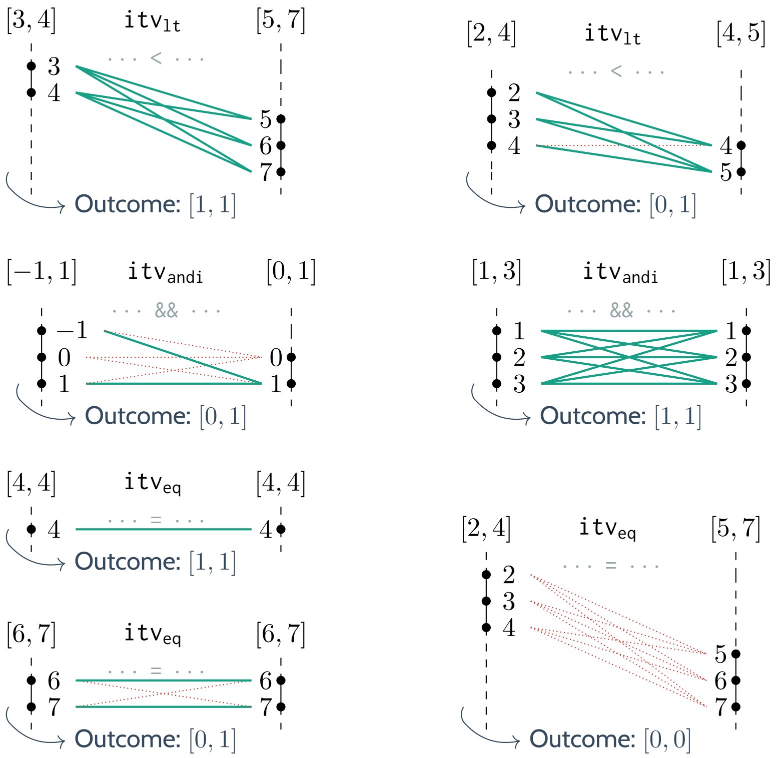

Fig. 3.10: Behavior of logical binary operations $\texttt{itv}_{\texttt{lt}}$, $\texttt{itv}_{\texttt{andi}}$ and $\texttt{itv}_{\texttt{eq}}$ on intervals. Every possible concrete boolean operation is represented by a line bewteen two numbers. A line is continuous and green when the operation returns `true`, and dotted red otherwise.

```
let β (x: int_m): itv = mk x x
let itv_eq (x y:itv): r:itv {(γ x = γ y) ⊆ γ r}
  = if x = y && itv_card x = 1
    then β 1 else if Bot? (x ⊓ y) then β 0 else mk 0 1
let itv_lt  (x y: itv): (r: itv {(γ x < γ y) ⊆ γ r})
  = match x, y with | Bot, _ | _, Bot → β 1
  | Val(|a,b|),Val(|c,d|) → if b<c then β 1
                                else if a>d then β 0 else mk 0 1
let itv_cγ  (i: itv) (x:int_m): r:bool {r ⟺ itv_γ i x}
  = match i with | Bot → false | Val (|l, u|) → l ≤ x && x ≤ u
type ubool = | Unk | TT | FF
let itv_as_bool (x:itv): ubool
  = if β 0=x || Bot?x then FF else if itv_cγ x 0 then Unk else TT
let itv_andi (x y: itv): (r: itv {(γ x && γ y) ⊆ γ r})
  = match itv_as_bool x, itv_as_bool y with
  | TT, TT  → β 1 | FF, _ | _, FF → β 0 | _, _ → mk 0 1
let itv_ori (x y: itv): (r: itv {(γ x `n_or` γ y) ⊆ γ r})
  = match itv_as_bool x, itv_as_bool y with
  | FF, FF → β 0 | TT, _ | _, TT  → β 1 | _, _ → mk 0 1
```

3.5.4 Backward Operators

While a forward analysis for expressions is essential, another powerful analysis can be made thanks to backward operators. Typically, it aims at extracting information from a test, and at refining the abstract values involved in this test, so as to gain in precision on those abstract values. As an example, consider the test x + y $\leq$ 5 in an abstract memory in which x is approximated by $[1,3]$, and y by $[3,6]$. As illustrated in Figure 3.11, if x is greater than 2 or y greater than 4, the test x + y $\leq$ 5 cannot be true. Thus, knowing that the test holds, we can refine the abstract of x to [1,3] and y to [3;4].

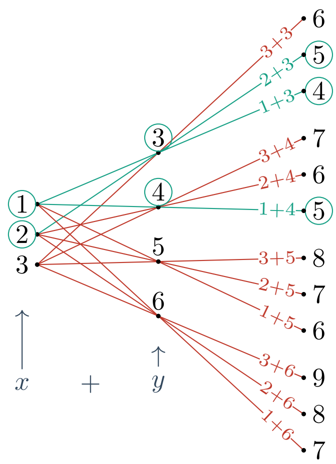

Fig. 3.11: Backward analysis of the expression x+y, knowing that x+y ∈ [-∞,5]. The initial abstraction for x is [1,3], the one for y is [3;5].

Given a concrete binary operator $\oplus$, we define $\overleftarrow{\oplus}$ its abstract backward counterpart. Assume given three intervals $x^\#$, $y^\#$, and $r^\#$. $\overleftarrow{\oplus}$ $x^\#$ $y^\#$ $r^\#$ tries to find the most precise intervals $x^{\#\#}$ and $y^{\#\#}$ supposing γ $x^\# \oplus \gamma$ $y^\#$ $\subseteq \gamma$ $r^\#$. The soundness of $\overleftarrow{\oplus}$ $x^\#$ $y^\#$ $r^\#$ can be formulated as below. We later generalize this notion of soundness with the type $\text{sound}_{\overleftarrow{op}}$, which is indexed by an abstract domain and a binary operation.

```
let x##, y## = (⊕⃖) x# y# r# in
 ∀ x y. (x ∈ γ x# ∧ y ∈ γ y# ∧ op x y ∈ γ r#)
   ⟹ (x ∈ γ x## ∧ y ∈ γ y##)
```

As the reader will discover in the rest of this section, this statement of soundness is proved entirely automatically against each and every backward operator for the interval domain. For op a concrete operator, $\text{sound}_{\overleftarrow{op}}$ itv op is inhabited by sound backward operators for op in the domain of intervals. If one shows that $\overleftarrow{\oplus}$ is of type $\text{sound}_{\overleftarrow{op}}$ itv (+), it means exactly that $\overleftarrow{\oplus}$ is a sound backward binary interval operator for (+). The rest of the listing shows how light in proof and OCaml-looking the backward operations. Below, we explain how $\overleftarrow{\mathtt{lt}}$ works: it is a bit complicated because it hides a "$\overleftarrow{\mathtt{ge}}$" operator.

```
let ←add: sound←op itv n_add = λx y r → x ⊓ (r-y), y ⊓ (r-x)
let ←sub: sound←op itv n_sub = λx y r → x ⊓ (r+y), y ⊓ (x-r)
let ←mul: sound←op itv n_mul = λx y r →
  let h (i j:itv) = (if j=β1 then i⊓r else i) in
  h x y, h y x
let ←eq:  sound←op itv n_eq
  = λx y r → match itv_as_bool r with
             | TT → x⊓y,x⊓y | _ → x,y
let (\) (x y: itv): (r: itv {(γ x \ γ y) ⊆ γ r}) =...
let ←and: sound←op itv n_and = λx y r →
  match itv_as_bool r, itv_as_bool x, itv_as_bool y with
  | FF, TT, _ → x, y ⊓ β 0        | FF, _, TT → x ⊓ β 0, y
  | TT,  _, _ → x \ β 0, y \ β 0 | _ → x, y
let ←or: sound←op itv n_or  = λx y r →
  match itv_as_bool r, itv_as_bool x, itv_as_bool y with
  | TT,FF,Unk | TT,FF,FF → x, y \ β 0 | TT,Unk,FF → x \ β 0, y
  | FF, _, TT | FF, TT, _ → x ⊓ β 0, y ⊓ β 0 | _ → x, y
```

Let us look at $\overleftarrow{\mathtt{lt}}$. Knowing whether `x < y` holds, $\overleftarrow{\mathtt{lt}}$ helps us to refine `x` and `y` to more precise intervals. Let `x` be the interval $[0; \mathtt{max}_{\mathtt{int_m}}]$, `y` be $[5; 15]$ and `r` be $[0; 0]$. Since the singleton $[0; 0]$ represents `false`, $\overleftarrow{\mathtt{lt}}$ `x y r` aims at refining `x` and `y` knowing that `x < y` doesn't hold, that is, knowing $\mathtt{x} \geq \mathtt{y}$. In this case, $\overleftarrow{\mathtt{lt}}$ finds $\mathtt{x'} = [5; \mathtt{max}_{\mathtt{int_m}}]$ and $\mathtt{y'} = [5; 15]$. Indeed, when `r` is $[0; 0]$, `itv_as_bool r` equals to **FF**. Then we rewrite $\neg(\mathtt{x} < \mathtt{y})$ either as $\mathtt{y} < \mathtt{x} + 1$ (when `x` is `incrementable`) or as $\mathtt{y} - 1 < \mathtt{x}$. In our case, the upper bound of `x` is $\mathtt{max}_{\mathtt{int_m}}$ (the biggest $\mathtt{int_m}$): `x` is not incrementable. Thus we rewrite $\neg([0; \mathtt{max}_{\mathtt{int_m}}] < [5; 15])$ as $[6; 16] < [0; \mathtt{max}_{\mathtt{int_m}}]$.

Despite the handling of these different cases, the implementation of $\overleftarrow{\mathtt{lt}}$ requires no proof: the SMT solver takes care of everything automatically.

```
let ←lt_true (x y: itv)
  = match x, y with | Bot, _ | _, Bot → x,y
  | Val⦇a,b⦈, Val⦇c,d⦈ → mk a (min b (d-1)), mk (max (a+1) c) d
let decrementable i=Val?i&&dfst(Val?.v i)>min_int_m  let incr.=...
let ←lt: sound←op itv n_lt
  = λx y r → match itv_as_bool r with | TT → ←lt_true x y
  | FF → if incrementable x // x < y ⟺ y > x+1
         then let ry, rx = ←lt_true y (itv_add x (β 1)) in
              itv_sub rx (β 1), ry
         else if decrementable y // x < y ⟺ y-1 > x
              then let ry, rx = ←lt_true (itv_sub y (β 1)) x in
                   rx, itv_add ry (β 1)
              else x,y | _ → x, y
```

3.6 A Word on Widening Operators

For simplicity, our abstract interpreter requires its abstract lattices to be equipped with a measure. Such measures exist only for lattices without infinite decreasing or increasing chains.

This section presents how Cousot's widening operators can be formalized in F*, and how they can be used to ensure convergence even in presence of lattice admitting infinite decreasing or increasing chains. A widening operator $\nabla : D^{\#} \to D^{\#} \to D^{\#}$ is a binary operator in an abstract domain $(D^{\#}, \subseteq)$ when:

- ∇ computes upper bounds, that is $\forall x\ y \in D^{\#}, x \subseteq x \nabla y \wedge y \subseteq x \nabla y$;
- for any sequence $(u_n)_{n\in\mathbb{N}}$ in $D^{\#}$, the sequence $(v_n)_{n\in\mathbb{N}}$ defined as $v_0 = u_0$, $v_{n+1} = v_n \nabla u_{n+1}$ stabilizes in finite time, that is $\exists n. v_n = v_{n+1}$.

3.6.1 An F* Definition

Below we define `ub (⪰)`, the type of binary operators computing upper bounds (⪰)-wise. `seq` τ defines sequences of values of type τ as maps from natural numbers to τ. The predicate `stabilizes s n` holds if the sequence `s` stabilizes after index `n`.

```
type ub #a ((⊆): order a) = x:a → y:a → r:a {x ⊆ r ∧ y ⊆ r}
type seq (t: Type) = ℕ → t
let stabilizes #t (s: seq t) (n:ℕ): prop
  = ∀ (i: ℕ). i ≥ n ⟹ s (i + 1) == s i
```

Given any sequence `u: seq` τ, `widen`$_{\text{seq}}$ `(∇) u` corresponds to the sequence $(v_n)_{n\in\mathbb{N}}$ described above, with ∇ the widening operator. The refinement type `wop (⊆)` is inhabited by widening operators, that is, binary operators computing upper bounds and ensuring convergence of $(v_n)_{n\in\mathbb{N}}$-like sequences. The function `stabilizes`$_{\text{at}}$ returns a witness of the index at which a sequence stabilizes; it is not computable, thus lives in the effect **GTot**.

```
let rec widen_seq (#(⊆):order τ) ((∇):ub (⊆)) (u:seq τ):seq τ
  = λn → if n=0 then u 0
              else widen_seq (∇) u (n-1) ∇ u n
type wop (#a: Type) ((⊆): order a)
  = w:ub (⊆){∀(s:seq a). ∃ n. stabilizes (widen_seq w s) n}
val stabilizes_at (s:seq τ{∃ n. stabilizes s n})
  : GTot (n:ℕ{stabilizes s n})
```

The function `finite_ub_to_widen (⊔) m` subtypes a join-like binary operator (⊔) into a widening operator, under the condition that a measure `m` exists. Most of the proof is carried out by the auxiliary lemma `finite_ub_to_widen' (⊔) m i u`, which is explained in Figure 3.12.

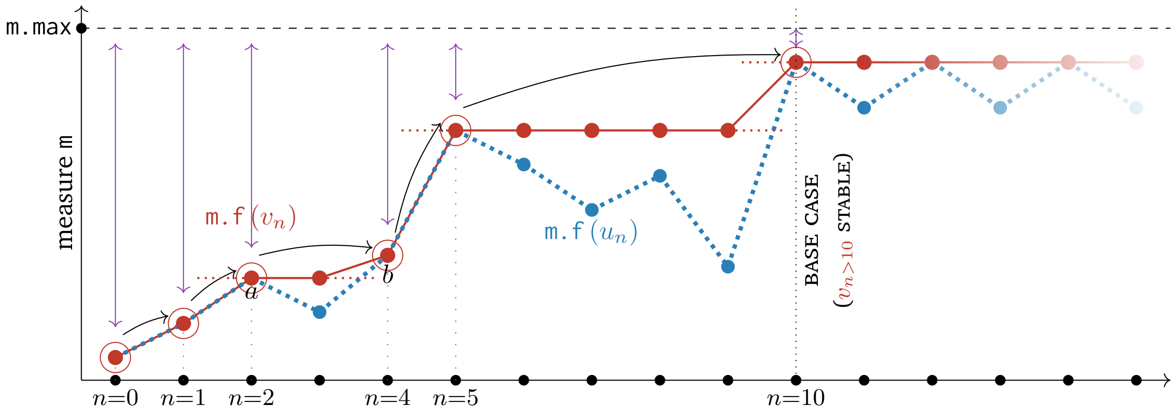

Fig. 3.12: Given a measure m and an arbitrary sequence u (in blue), v (in red) is the sequence widen$_{\text{seq}}$ (⊔) u, with a join operator. By construction, v is monotonically increasing. The proof that (⊔) is a widening operator consists in an induction on i a growing index, that makes the size of the purple arrows decrease. These arrows represent the difference between the maximal measure and the measure for v_i. On the illustration, when i=8, we hit the base case (hatched area on the right): $v_j = v_{j+1}$ for any j greater than 8. Otherwise, the sequence v is not stable after i: there exists a j so that $v_i \neq v_j$. For instance here, if $i = 2$, there exists $j = 4$ with $v_2 \neq v_4$. Thus, we use our hypothesis of recurrence: there exists an index so that v stabilizes, by calling our lemma recursively with $i = j$ (the arrow from a to b on the illustration).

```
let increasing ((⊆):order τ) (u:seq τ) = ∀i. u i ⊆ u (i+1)
let rec increasing_transitive ((⊆):order τ) (u:seq τ) (i j:ℕ)
  : Lemma (requires increasing (⊆) u ∧ i<j)
          (ensures  u i ⊆ u j) (decreases j-i)
  = if j-i=1 then () else increasing_transitive (⊆) u i (j-1)
let rec finite_ub_to_widen'
  (#(⊆): order τ) ((⊔): ub (⊆))
  (m: measure (⊆)) (i: ℕ) (u: seq τ)
  : Lemma (ensures ∃ n. stabilizes (❶ widen_seq (⊔) u) n)
          (decreases m.max - m.f (❷ widen_seq (⊔) u i))
  = let v = ❸ widen_seq (⊔) u in
    let p (j:ℕ) = ❹ j > i ∧ v i =/= v j in
    let goal = ∃ n. stabilizes (widen_seq (⊔) u) n in
    let sub (): Lemma (requires ∃ j. p j) (ensures goal)
      = ∃_elim goal () (λ(❺j: ℕ{p j}) →
                increasing_transitive (⊆) v i j;
                finite_ub_to_widen' (⊔) m j u)
    in move_requires sub ()
let finite_ub_to_widen
  (#o: order τ) (w: ub o) (m: measure o): wop o
  = forall_intro (finite_ub_to_widen' w m 0); w
```

3.6.2 Computing Fixpoints for Abstract Operators

Our ultimate goal is to be able to infer loop invariant. Consider the loop **Loop** f, and let the abstract operator f$^\#$: **A**→**A**, with **A** being an abstract domain, ordered by ⊆.

We are looking for an approximation p:**A** for any number of iteration of f$^\#$. Both seq$_{\text{wop}}$ (∇) f$^\#$ x$_0$ and seq$_{\text{wop}}$' (∇) f$^\#$ x$_0$ implement the sequence $v_0 = \texttt{x}_0$, $v_{n+1} = v_n \nabla f^\# v_n$. Their equivalence is proven by seq_wop_eq. The latter defines $(v_n)_{n\in\mathbb{N}}$ in terms of seq$_{\text{wop}}$'' (∇) f$^\#$ x$_0$ and widen$_{\text{seq}}$, so that it benefits from widen$_{\text{seq}}$ stabilization. This stabilization allows us to define fp so that fp (∇) f$^\#$ x$_0$ terminates.

```
let rec seq_wop (#o: order τ) (w: wop o) (f:τ→τ) (x0:τ) (n:ℕ)
  : Tot τ (decreases n)
  = if n=0 then x0 else let x1=seq_wop w f x0 (n-1) in
                         x1 `w` f x1
let rec seq_wop'' (#o:order τ) (w:wop o) (f:τ→τ) (x0:τ) (n:ℕ)
  : Tot τ (decreases n)
  = if n=0 then x0
    else f (widen_seq w (λm → if m≥n then x0
                              else seq_wop'' w f x0 m)
                         (n-1))
let seq_wop' (#o: order τ) (w: wop o) (f:τ → τ) (x0: τ)
  = widen_seq w (seq_wop'' w f x0)
val seq_wop_eq (#o:order τ) (w:wop o) (f:τ→τ) (x0:τ) (n:ℕ)
  : Lemma (seq_wop w f x0 n == seq_wop' w f x0 n)
val fp (#o: order τ) (w: wop o) (f:τ → τ) (x0: τ)
  : x:τ{∃m. x == seq_wop' w f x0 m ∧ stabilizes (seq_wop w f x0) m}
```

3.7 Specialized Abstract Domains

Abstract domains are defined in Section 3.4 as lattices equipped with a sound concretization operation. Our abstract interpreter analyses IMP programs: its expressions are numerical, and IMP is equipped with a memory. Thus, this section defines two specialized abstract domains: one for numerical abstractions, and another one for memory abstractions.

3.7.1 Numerical Abstract Domains

In Section 3.5.4, we explain what a sound backward operator is in the case of the abstract domain of intervals. There, we mention a more generic type $\text{sound}_{\overleftarrow{op}}$ that states soundness for such operators in the context of any abstract domain. We present its definition below:

```
type sound_←op (a:Type) {|l:adom a|} (op:l.c→l.c→l.c)
  = ←op: (a → a → a → (a & a)) {
      ∀ (x# y# r#: a). let x##, y## = ←op x# y# r# in
        (∀ (x y: l.c). (x ∈ γ x# ∧ y ∈ γ y# ∧ op x y ∈ γ r#)
                 ⟹ (x ∈ γ x## ∧ y ∈ γ y##))}
```

We define the specialized typeclass num_{adom} for abstract domains that concretize to machine integers. A type that implements an instance of num_{adom} should also have an instance of adom, with int_m as concrete type. Whence the fields na_{adom}, and adom_{num}. Moreover, we require a computable concretization function cgamma, that is, a function that maps abstract values to computable sets of machine integers: int_m → **bool**. The β operator lifts a concrete value in the abstract world. We also require the abstract domain to provide both sound forward and backward operator for every syntactic operator of type binop presented in Section 3.2. The function abstract_binop maps an operator op of type binop to a sound forward abstract operator. Its soundness is encoded as a refinement. Similarly,

$\overleftarrow{\texttt{abstract\_binop}}$ maps a `binop` to a corresponding sound backward operator. To ease backward analysis, $\texttt{gt}_0$ and $\texttt{lt}_0$ are abstractions for non-null positive and negative integers.

```
class num_adom (a: Type) =
{ na_adom: adom a; adom_num: squash (na_adom.c == int_m)
; cgamma: x#:a → x:int_m → b:bool {b ⟺ x ∈ γ x#}
; abstract_binop: op:_→ i:a → j:a → r:a {lift op (γ i) (γ j) ⊆ γ r}
; ←abstract_binop: (op: binop) → sound_←op a (concrete_binop op)
; gt_0: x#:a {∀(x:int_m). x>0 ⟹ x ∈ γ x#}
; lt_0: x#:a {∀(x:int_m). x<0 ⟹ x ∈ γ x#}; β: x:int_m → r:a{x ∈ γ r} }
```

For a proposition `p`, the F* standard library defines `squash p` as the type `_:unit{p}`, that is, a refinement of the unit type. This can be seen as a lemma with no parameter.

3.7.1.1 Instance for intervals

Section 3.5 defines everything that is required by $\texttt{num}_{\texttt{adom}}$, thus below we give an instance of the typeclass $\texttt{num}_{\texttt{adom}}$ for intervals.

```
instance itv_num_adom: num_adom itv = {
  na_adom = solve; adom_num = (); cgamma = itv_cγ ; β = (λ x → β x);
  abstract_binop = (function | Plus → itv_add ... | Or → itv_ori);
  ←abstract_binop = (function | Plus → ←add ... | Or → ←or );
  lt_0 = (mk min_int_m (-1)); gt_0 = (mk ( 1) max_int_m) }
```

3.7.2 Memory Abstract Domains

From the perspective of IMP statements, an abstract domain for abstract memories is fairly simple. An abstract memory should be equipped with two operations: assignment and assumption. Those are directly related to their syntactic counterpart **Assume** and **Assign**. Thus, $\texttt{mem}_{\texttt{adom}}$ has a field `assume_` and a field `assign`. The correctness of these operations are elegantly encoded as refinement types.

Let us explain the refinement of `assume_`: let $\texttt{m}_0^\#$ an abstract memory, and `e` an expression. For every concrete memory $\texttt{m}_0$ abstracted by $\texttt{m}_0^\#$, the set of acceptable final memories $\texttt{osem}_{\texttt{stmt}}$ (**Assume** e) $\texttt{m}_0$ should be abstracted by `assume_` $\texttt{m}_0^\#$ `e`.

```
class mem_adom μ = { ma_adom: adom μ; ma_mem: squash (ma_adom.c == mem);
  assume_: m0#:μ → e:expr → m1#:μ
    {∀ (m0: mem{m0 ∈ γ m0#}). osem_stmt (Assume   e) m0 ⊆ γ m1#};
  assign: m0#:μ → v:varname → e:expr → m1#:μ
    {∀ (m0: mem{m0 ∈ γ m0#}). osem_stmt (Assign v e) m0 ⊆ γ m1#}}
```

3.8 A Weakly-Relational Abstract Memory

In this section, we define a weakly-relational abstract memory. This abstraction is said weakly-relational because the entrance of an empty abstract value in the map systematically launches a reduction of the whole map to **Bot**. Below we define an abstract memory (amem) as either an unreachable state (**Bot**), or a mapping (map τ) from varname to abstract values τ. The mappings map τ are equipped with the utility functions mapi, map_1, map_2 and fold.

```
type map τ =...                          type amem τ = withbot (map τ)
let get': map τ → varname → τ =...       let fold: (τ→τ→τ) → map τ → τ =...
let mapi: (varname → τ → β) → map τ → map β =...
let map₁: (τ→β) → map τ → map β = λf → mapi (λ_ → f)
let map₂: (τ→β→γ) → map τ → map β → map γ =...
```

3.8.1 A Lattice Structure

The listing below presents amem instances for the typeclasses order, lattice and mem_{adom}. Once again, the various constraints imposed by these different typeclasses are automatically discharged by the SMT solver.

```
let amem_update (k: varname) (v: τ) (m: amem τ): amem τ
  = match m with | Bot → Bot
    | Val m → Val (mapi (λk' v' → if k'=k then v else v') m)
instance amem_lat {| l: adom τ |}: lattice (amem τ) =
  { corder = withbot_ord (λm₀ m₁ → fold (&&) (map₂ corder m₀ m₁))
  ; join = (λx y → match x, y with
      | Val x, Val y → Val (map₂ join x y) | m,Bot | _,m → m)
  ; meet = (λx y → match x, y with
      | Val x, Val y →
        let m = map₂ (⊓) x y in
        if fold (||) (mapi (λ_ v → l.adom_lat.corder v bottom) m)
        then Bot else Val m
      | _ → Bot); bottom = Bot; top = ...}
instance amem_adom {|l:adom τ|}: adom (amem τ) = { c = mem' l.c
  ; adom_lat=solve; meet_law=(λ_ _→()); top_law=(λ_→()); bot_law=(λ_→())
  ; γ = withbot_γ (λm# m → fold (∧) (mapi (λv x → m v ∈ γ x) m#))
  ; widen = (λx y → match x, y with
    | Val x, Val y → Val (map₂ widen x y) | m,Bot | _,m → m)
  ; order_measure = let {max; f} = l.order_measure in
    { f = (function | Bot → 0 | Val m# → 1 + fold (+) (map₁ f m#))
    ; max = 1 + max × 4 }}
```

The rest of this section defines a mem_{adom} instance for our memories amem. The typeclass mem_{adom} is an essential piece in our abstract interpreter: it provides the abstract operations for handling assumes and assignments.

3.8.2 Forward Expression Analysis

We define $\mathtt{asem}_{\mathtt{expr}}$, mapping expressions to abstract values of type τ. It is defined for any abstract domain, whence the typeclass argument $\{|\mathtt{num}_{\mathtt{adom}}\ \tau|\}$. The abstract interpretation of an expression e given $\mathtt{m}^\#_\emptyset$ an initial memory is defined below as $\mathtt{asem}_{\mathtt{expr}}\ \mathtt{m}^\#_\emptyset$ e. It is specified via a refinement type to be a sound abstraction of the operational semantics $\mathtt{osem}_{\mathtt{expr}}\ \mathtt{m}_0$ e of e. This function leverages the operators from the different typeclasses for which we defined instances just above. β:$\mathtt{int}_{\mathtt{m}}\to\tau$ and abstract_binop:binop→... come from $\mathtt{num}_{\mathtt{adom}}$, while top:$\tau$ comes from lattice.

```
val get: m:amem τ {Val? m}→varname→τ
let get (Val m) = get' m
let rec asem_expr {|num_adom τ|} (m#_∅: amem τ) (e: expr)
  : (r: τ { ∀ (m_0: mem). m_0 ∈ γ m#_∅ ⟹ osem_expr m_0 e ⊆ γ r })
  = if m#_∅ ⊑ bottom then bottom else
    match e with
    | Const x → β x | Unknown → top | Var v → get m#_∅ v
    | BinOp op x y → abstract_binop op (asem_expr m#_∅ x)
                                       (asem_expr m#_∅ y)
```

3.8.3 Backward Analysis

Our aim is to have an instance for our memory of $\mathtt{mem}_{\mathtt{adom}}$: it expects an assume_ operator. Thus, below a backward analysis is defined for expressions. Given an expression e, an abstract value $r^\#$ and a memory $\mathtt{m}^\#_\emptyset$, $\overleftarrow{\mathtt{asem}}$ e $r^\#$ $\mathtt{m}^\#$ computes a new abstract memory. That abstract memory refines the abstract values held in $\mathtt{m}^\#_\emptyset$ as much as possible under the hypothesis that e lives in $r^\#$. The soundness of this analysis is encoded as a refinement on the output memory. Given any concrete memory $\mathtt{m}_0$ and integer v approximated by some $r^\#$, if the operational semantics of e at memory $\mathtt{m}_0$ contains v, then $\mathtt{m}_0$ should also be approximated by the output memory.

When e is a constant which is not contained in the concretization of the target abstract value $r^\#$, the hypothesis "e lives in $r^\#$" is false, thus we translate that fact by outputting the unreachable memory bottom. In opposition, when e is **Unknown**, the hypothesis does not bring any new knowledge, thus we return the initial memory $\mathtt{m}^\#_\emptyset$. In the case of a variable lookup (i.e. e = **Var** v for some v), we consider $x^\#$, the abstract value living at v. Since our goal is to craft the most precise memory such that **Var** v is approximated by $r^\#$, we alter $\mathtt{m}^\#_\emptyset$ by assigning $x^\# \sqcap r^\#$ at the variable v. Finally, in the case of binary operations, we make use of the backward operators and of recursion. Figure 3.13 illustrates how the recursive backward analysis is performed. Note that it is the only place where we need to insert a hint for the SMT solver: we assert an equality by asking F* to normalize the terms. We explicitly state that the operational semantics of a binary operation reduces to two existentials: we manually unfold the definition of $\mathtt{osem}_{\mathtt{expr}}$ and lift_binop. The **decreases** clause explains to F* why and how the recursion terminates.

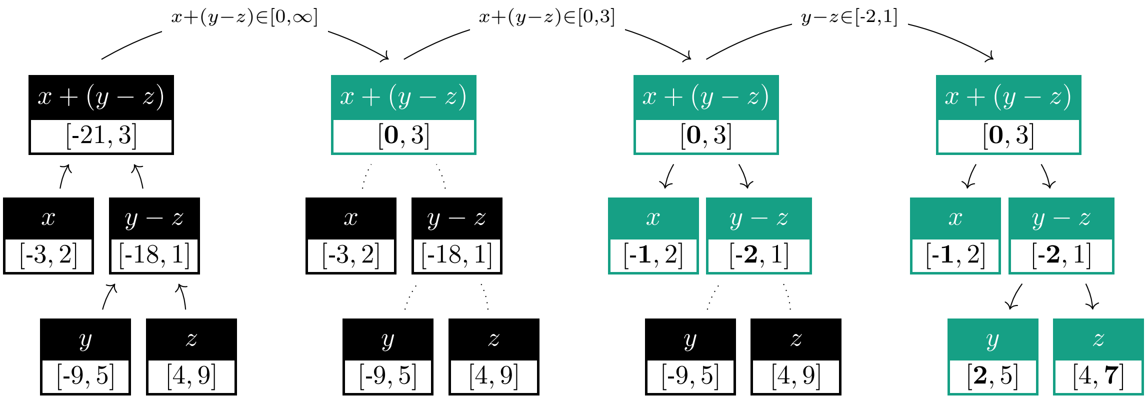

Fig. 3.13: Example of backward analysis for the expression $x+(y-z)$ given the hypothesis that it is positive. The initial abstractions are $[-3,2]$ for x, $[-9,5]$ for y and $[4,9]$ for z.

```
let rec ⟵asem {|l:num_adom τ|} (e: expr) (r#: τ) (m0#: amem τ)
: Tot (m1#: amem τ { (* decreases: *) m1# ⊑ m0# ∧ (* soundness: *)
      (∀(m0:mem) (v:int_m). (v∈γ r# ∧ m0∈γ m0# ∧ v ∈ osem_expr m0 e)
                                        ⟹ m0 ∈ γ m1#)}) (decreases e)
  = if m0# ⊑ bottom then bottom else match e with
  | Const x → if cgamma r# x then m0# else bottom | Unknown → m0#
  | Var v→let x#: τ= r# ⊓ get m0#v in
          if x#⊑bottom then Bot else amem_update v x# m0#
  | BinOp op ex ey → let ⟵op = ⟵abstract_binop op in
      let x#, y# = ⟵op (asem_expr m0# ex) (asem_expr m0# ey) r# in
      let r#: amem τ = ⟵asem ex x# m0# ⊓ ⟵asem ey y# m0# in
      assert_norm (∀ (m: mem) (v: int_m). v ∈ osem_expr m e
        ⟺ (∃ (x y:int_m). x ∈ osem_expr m ex ∧ y ∈ osem_expr m ey
                              ∧ v == concrete_binop op x y));
      r#
```

3.8.4 Iterating the Backward Analysis

While a concrete test is idempotent, it is not the case for abstract ones. Our goal is to refine an abstract memory under a hypothesis as far as possible. Since ⟵asem is proven sound and decreasing, we can repeat the analysis as much as we want. We introduce `prefixpoint` that computes a pre-fixpoint. However, even if the function from which we want to get a prefixpoint is decreasing, this is not a guarantee for termination. The type `measure` below associates an order to a measure that ensures termination. Such a measure cannot be implemented for a lattice that has infinite decreasing or increasing chains. We also require a maximum for this measure, so that we can reverse the measure easily in the context of postfixpoints iteration.

```
type measure #a (ord: a → a → bool)
  = { f: f: (a → ℕ) {∀ x y. x `ord` y ⟹ x ≠ y ⟹ f x < f y}
    ; max: (max: ℕ {∀ x. f x < max}) }
```

Let us focus on prefixpoint: given an order ⊑ with its measure m, it iterates a decreasing function f, starting from a value x. The argument r is a binary relation which is required to hold for every couple (x, f x). r is also required to be transitive, so that (morally, for any n) r x (f^n x) holds. prefixpoint is specified to return a prefixpoint y, that is, with r x y holding.

```
let rec prefixpoint ((⊑): order τ) (m: measure (⊑))
  (r: τ→τ→prop {trans r}) (f: τ→τ {∀e. f e ⊑ e ∧ r e (f e)}) (x:τ)
  : Tot (y: τ{r x y ∧ f y == y ∧ y ⊑ x}) (decreases (m.f x))
  = let x' = f x in if x ⊑ x' then x else prefixpoint (⊑) m r f x'
```

Below is defined $\overleftarrow{\text{asem}}$_fp, the iterated version of $\overleftarrow{\text{asem}}$. Besides using prefixpoint, the only thing required here is to spell out t, the relation we want to ensure.

```
let asem_fp {|num_adom τ|} (e:expr) (r:τ) (m#_0:amem τ)
  : Tot (m#_1: amem τ {(∀ (m_0:mem) (v:int_m). m#_1⊑m#_0 ∧
                  (v∈γr ∧ m_0∈γm#_0 ∧ v∈osem_expr m_0 e) ⟹ m_0∈γm#_1)})
  = let t (m#_0 m#_1: amem τ) = ∀ (m: mem) (v: int_m).
      (v ∈ γ r ∧ m ∈ γ m#_0 ∧ v ∈ osem_expr m e) ⟹ m ∈ γ m#_1 in
    prefixpoint corder order_measure t (asem e r) m#_0
```

3.8.5 A mem$_{\text{adom}}$ Instance

We defined both a forward and backward analysis for expressions. Implementing a mem$_{\text{adom}}$ instance for amem is thus easy, as shown below. For any numerical abstract domain τ, amemory_mem_adom provides a mem$_{\text{adom}}$, that is, an abstract domain for memories, providing nontrivial proofs of correctness. Still, this is proven automatically.

```
instance amemory_mem_adom {| nd: num_adom τ |}: mem_adom (amem τ) =
  let adom: adom (amem τ) = amem_adom in { ma_adom = adom; ma_mem = ()
  ; assume_ = (λm# e → asem_fp e gt_0 m# ⊔ asem_fp e lt_0 m#)
  ; assign = (λm# v e → let v#: τ = asem_expr m# e in
                 if v# ⊑ bottom then Bot else amem_update v v# m#)}
```

3.9 Statement Abstract Interpretation

Wrapping up the implementation of our abstract interpreter, this section presents the abstract interpretation of IMP statements. For every memory type μ that instantiates the typeclass of abstract memories mem$_{\text{adom}}$, the abstract semantics asem$_{\text{stmt}}$ maps statements and initial abstract memories to final memories. mem$_{\text{adom}}$ is defined and proven sound below.

Given a statement s, and an initial abstract memory $m^\#_0$, mem$_{\text{adom}}$ s $m^\#_0$ is a final abstract memory so that for any initial concrete memory m approximated by $m^\#_0$ and for any acceptable final concrete memory m' considering the operational semantics, `m'` is approximated by `mem_adom s` $m_0^\#$. Here, we give two hints to the SMT solver: by normalization (`assert_norm`), we unfold the operational semantics in the case of choices or sequences. The analysis of an assignment or an assume is very easy since we already have operators defined for these cases. The sequence of two statements is handled by recursion. Similarly, when the statement is a choice, we recurse on its two subterms, and merge together the two resulting abstract memories. The last case to be handled is the loop, that is some statement of the shape `Loop body`. We compute a fixpoint $m_1^\#$ for body, by widening: it therefore approximates correctly the operational semantics of `Loop body`, since it is defined as a transitive closure. The standard library of F* [FStdLib] provides the lemma `stable_on_closure`; of which we give a simplified signature below. The concretization γ $m_1^\#$ is a set, that is a predicate: we use this lemma with γ $m_1^\#$ as predicate `p` and with the operational semantics as relation `r`.

```
val simplified_stable_on_closure: r:(τ → τ → prop) → p:(τ → prop)
  → Lemma (requires ∀ x y. p x ∧ r x y ⟹ p y)
          (ensures ∀ x y. p x ∧ closure r x y ⟹ p y)

let rec asem_stmt {| md: mem_adom μ |} (s: stmt) (m#_0: μ)
  : (m#_1:μ{∀(m m':mem). (m∈γ m#_0 ∧ m'∈osem_stmt s m) ⟹ m'∈γ m#_1})
  = assert_norm(
      (∀s0 s1 (m0 mf:mem). osem_stmt (Seq s0 s1) m0 mf
        == (∃(m1:mem). m1 ∈ osem_stmt s0 m0 ∧ mf ∈ osem_stmt s1 m1))
    ∧ (∀a b (m0 mf:mem). osem_stmt (Choice a b) m0 mf
        == (mf ∈ (osem_stmt a m0 ∪ osem_stmt b m0)))
    );
    if m#_0 ⊑ bottom then bottom
    else match s with             | Assign v e → assign m#_0 v e
    | Assume e → assume_ m#_0 e | Seq s t → asem_stmt t (asem_stmt s m#_0)
    | Choice a b → asem_stmt a m#_0 ⊔ asem_stmt b m#_0
    | Loop body→let m#_1: μ = postfixpoint corder order_measure
                  (λ(m#:μ) → widen m# (asem_stmt body m# <: μ))
                in stable_on_closure (osem_stmt body) (γ m#_1) (); m#_1
```

Below we show the definition of `postfixpoint` that is similar to `prefixpoint`. However, it is simpler because it only ensures that its outcome is a postfixpoint.

```
let rec postfixpoint ((⊑): order τ) (m: measure (⊑))
  (f: τ → τ {∀ x. x ⊑ f x}) (x: τ)
  : Tot (y: τ{f y == y ∧ (⊑) x y}) (decreases (m.max - m.f x))
  = let x' = f x in if x' ⊑ x then x else postfixpoint (⊑) m f x'
```

3.10 Related work

Efforts in verified abstract interpretation are numerous, and most of them have been focused on concretization-based formalizations. Such formalizations aim at providing provably sound and terminating abstract interpreter implementations. This concretization-based approach has been proven successful [Dav05; CP10; BLP16], and scales up to Verasco [Jou+], a real-world abstract interpreter verified in Coq.

Verasco targets C#minor, one of the intermediary languages the formally verified C compiler CompCert [Ler+16] uses. Since the semantics preservation theorem of CompCert guarantees that properties on C#minor semantics carry over to their compiled assembly code counterpart, Verasco's analysis also carries out to assembly code. Blazy et al. [BLP16] and Verasco closely follow the modular design of Astrée [Cou+05]. Their design consists in three layers which are interconnected with clear interfaces: numerical abstract domains, memory models and the abstract iterator itself. Verasco implements both non-relational abstract domains (integer intervals, integer congurences, floating-point intervals, points-to sets) and relational ones (convex polyhedrons, symbolic equalities, octagons [Jou17]). Our interpreter is an order of magnitude simpler, but still follows this modular architecture.

Such analysers however require a non-trivial amount of mechanized proofs; in constrast, this chapter shows that implementing a formally verified abstract interpreter with very few manual proofs is possible. Ours is up to 17 times more proof efficient. It is very compact, and requires a negligible amount of manual proofs. Table 3.1 compares the line-of-proof vs. line-of-code ratio of our implementation compared to some of the available verified abstract interpreters. This comparison has its limits, since the different formalizations do not target the same programming languages: [Jou+] and [BLP16] handle the full C language, while [CP10] and the current paper deal with more simple imperative languages. Also, proof effort does not usually scale linearly.

	Code	Proof	Ratio	Feature set
This paper	487	39	0.08	Simple imperative language
Pichardie et al. [CP10]	3 725	5 020	1.35	Simple imperative language
Verasco [Jou+]	16 847	17 040	1.01	CompCert C langage
Blazy et al. [BLP16]	4 000	3 500	0.87	CompCert C langage

Table 3.1: Comparison of the number of *line of code* and of *line of proof* of different sound abstract interpreters.

The work of Nipkow [Nip12] is similar to ours in terms of objectives: formalizing a sound abstract interpreter in a comprehensible way. The originality of this work is to iterate over ASTs annotated with semantics information. This approach yields a very pleasant and illustrative abstract interpretation: as the iteration goes, one can observe the annotations getting more precise. This iterating process is proven sound and implemented in about 2000 lines of code using the Isabelle/HOL [NPW02] proof assistant. Since Nipkow aims at simplicity, its abstract interpreter has no full fledged widening or narrowing, consider a memory of unbounded integer, and only consider addition as arithmetic operation. Still, our interpreter is about four times more compact. Similarly, Bertot [Ber08] also presents an annotation-based approach. Its particularity is that the soundness of the abstract analysis is stated and proven against a weakest-precondition semantics.

The work of Darais et al. [DMV15; Dar+17] advocates for using Galois connections to prove abstract interpreter implementations sound. [DMV15] aims at a reusable and modular abstract interpretation. While Verasco is modular in the sense of, e.g., its abstract domains, in the work of Darais, the aim is to implement a meta-theory for sound abstract interpretations in a language-agnostic way. This is achieved by defining the concept of *Galois transformers*, which are monad transformers that transport Galois connection properties. A Galois transformer provides a given static analysis, and is proven sound once for all, independently from the language. In this framework, an abstract analyzer is an interpreter of computations whose monad is a stack of Galois transformers, each providing a specific analysis. One of the benefits of Galois transformers is that they are self-contained in terms of soundness: the hope is that this approach is therefore more comprehensible to a public unfamiliar with strong typing or proof assistants. Our work makes formally verified abstract interpretation more accessible in a much more pragmatic way. Our soundness statements are very straightforward and standard in terms of abstract interpretation. Our implementation being specifically designed with F* automations in mind, it yields almost no manual proving.

3.11 Conclusion

We presented almost the entire code of our abstract interpreter for IMP. Our approach to abstract interpretation is concretization-based, and follows the methodology of [BLP16; Jou+]. While using F*, we did not encounter any issue regarding expressiveness, and additionally gained a lot in proof automatization, to finally implement a fairly modular abstract interpreter. The sources of our abstract interpreter sources are available along with a set of example programs; building it natively or as a web application is easy, reproducible[3] and automated.

[3] Our build process relies on the purely functional Nix package manager.

This work is very far from the scope of Verasco which required about four years of human time [Lap15; Jou16], but our results, which required 3 months of work with F* expertise, are very encouraging.

As further work, we aim at following the path of Verasco by adding real-world features to our abstract interpreter. It would be interesting to study how much manual proving is necessary to implement more powerful abstract domains (e.g. octagons domains). Also, we would like to consider a more realistic target language such as one of the CompCert C-like input languages. One of the weaknesses of Verasco is its poor efficiency: using purely functional data structures and Coq's integer arithmetic, Verasco takes a lot of time to analyze programs. Using the C DSL Low* (See Section 2.3.3), it would be possible to write an abstract interpreter which is both very efficient and formally verified. Of course such an efficient analyzer would come with a nontrivial additionnal effort related to Low* and low-level data structures and related invariants.

This development also opens the way to enrich F* automation via verified abstract interpretation. This is exactly what the following Chapter 4 investigates.

CHAPTER 4

Abstract Interpretation as a Weakest-Precondition Monad Transformer

> With great powers comes great annotations.

Contents

F* is a dependently typed programming language, just like Coq, Agda or Idris. As illustrated in Chapter 2, type systems in such languages are very expressive. By contrast with most verification approach, a dependently typed programming language sets on equal footing (i) programs, (ii) specifications and (iii) proofs, leaving a very thin frontier between them. Specification can leverage program facilities (i.e. higher-order functions, polymorphism, etc.), programming can be driven by specifications and eased by proof facilities (i.e. tactics), etc. The dependent type approach is thus far from being only focused on verification: this approach also revolutionizes the experience of programming [Bra17]. It moreover allows one to place the cursor from simple types ensuring no runtime failure to arbitrarily rich types, encoding precise program specifications.

The tremendous power brought by dependent types comes with the cost of undecidability. Since types can virtually encode any property, no procedure deciding whether an arbitrary term inhabits an arbitrary type exists. The type system of F* relies in part on an SMT solver to decide whether a given term inhabits a certain type, so that the F* user hits the undecidability of the type system less often. When the type system cannot decide, the programmer has to supply evidences to F*: add annotations or lemmas (See Section 2.1.3), write a tactic-based proof (See Section 2.3.2), etc.

Inference of refinement types The type system of F* has great automation for deciding type inhabitation, but what about type inference? In the case of recursion, inferring precise types is difficult; non-trivial recursion requires annotations in F*. Consider the recursive functions add_0, add_1 and add_2 below:

```
1  int i = 0;  int lst[5] = {-1, -2, 42, -3, 5};
2  while (i<5 && lst[i]<0)  i++;     // i ∈ [0;4] ∧ lst[i] < 0
3  if (i >= 5) failwith("Not found");
4  else return lst[i];               // i ∈ [0;4] ∧ lst[i] ≥ 0
```

Fig. 4.1: Example *find* program, implemented in a C-like language.

```
let rec add₀ (x y: ℕ)                 = if x=0 then y else 1 + add₀ (x - 1) y
        ↪ has type ℕ → ℕ → ℤ
let rec add₁ (x y: ℕ):   ℕ            = if x=0 then y else 1 + add₁ (x - 1) y
        ↪ has type ℕ → ℕ → ℕ
let rec add₂ (x y: ℕ): r:ℕ {r = x + y} = if x=0 then y else 1 + add₂ (x - 1) y
        ↪ has (dependent) type x:ℕ → y:ℕ → r:ℕ {r = x + y}
```

The result type of add_0 is omitted; F* infers $\mathbb{Z}$. The operator (+) is of type $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, thus the expression add_0 (x - 1) y has to be *at least* of type $\mathbb{Z}$. This type is weak: for instance, F* is not able to automatically subtype add_0 16 2 into a natural number. The fact the type system of F* is undecidable is here irrelevant: add_1, that simply adds a type annotation, is type-checked automatically. One step further, the return type of add_2 embeds a very precise specification. In fact, the refinements added by add_1 and add_2 are recursion invariants: it is not surprising F* type system doesn't infer such precise invariants.

Abstract interpretation As presented in Chapter 3, abstract interpretation is precisely good at inferring invariants: an abstract interpreter statically analyses a program to *discover* properties and invariants automatically. The expressiveness of an abstract interpreter –even with numerous abstract domains– is limited. For instance, the invariant spelled out in add_1 is easily represented in an abstract domain (i.e. sign or interval domain), while the one of add_2 is too complex for simple abstract domains. This lack of expressiveness is fine: the work presented in this chapter aims at freeing the F* programmer from *boring* annotations. By *boring*, we mean trivial, simple or repetitive properties.

Example The imperative program in Figure 4.1 gives us an additional motivation: imperative programs often yield verbose annotations. *find* simply finds a positive number in a list of values. Line 2 contains the loop invariant and line 4 the post-condition. In a standard weakest-precondition approach, adding the framed part is mandatory. Our technique provides a hybrid weakest-precondition calculus in which the user is freed from this task. While this example is very simple, it reveals some of the many small annotations a typical low-level F* program would require.

Contributions This chapter provides a weakest-precondition calculus transformer indexed by sound abstract interpreters. The resulting, so called hybrid, weakest-precondition calculus embeds an abstract interpreter that inserts automatically sound invariants as free hypothesis. We define a hybrid weakest-precondition calculus enriched by abstract interpretation on a minimalistic imperative language (Sections 4.3 and 4.4). We provide a statement of the soundness for the generated hybrid proof obligations along with a proof partially mechanized in F⋆ (Section 4.5). As artifact, we provide an implementation of our transformer in F⋆, parameterized over a given abstract interpreter, and we instantiate it. Section 4.7 explores the generalization and mechanization of this hybridization, by turning our experiment into an actual monad transformer.

Current limitations. While our approach is indeed able to generate lighter and sound proof obligations, our hybridization currently yields an exponential number of forks of abstract analysis as the number of conditional statements increases. This forking issue for conditionals has repercussions on fixpoint iterations as well. The Section 4.4.4 gives more details and explanations about this.

4.1 Overview

This section provides a glimpse of our approach by expressing the *find* program of Figure 4.1 in the subset language of F⋆, Low⋆ (see Section 2.3.3). At ❶ and ❷, in Figure 4.2 we allocate an array `l` of integers `{-1,-2,42,-3,5}`. Following Low⋆'s syntax, literals with suffix `ul` are **u**nsigned integer **l**iterals; those with suffix `l` are signed literals. At ❸, Low⋆'s operator `v` converts a signed integer to a natural number. The while loop at ❹ looks for a positive number. `test` is the condition of the while loop, `body` its body that increments `i`. $\texttt{test}_{\texttt{pre}}$ and $\texttt{test}_{\texttt{post}}$ are the pre- and post-conditions of function `test`. It returns `true` when `i` is in the bounds of `l` and points to a negative number. At ❺, `main` either finds a positive number and returns it, or throws an error.

The Effect ST Function `main` has a particular type signature: **`ST`** $\texttt{int}_{32}$ $(\lambda \mathsf{h}_0 \rightarrow \top)$ $(\lambda \mathsf{h}_0\ \ \mathsf{r}\ \ \mathsf{h}_1 \rightarrow \mathsf{r} \geq \texttt{0l})$, which defines it as an effectful computation producing an $\texttt{int}_{32}$. The effect **`ST`** is a variant of the effect **`Stack`** presented in Figure 2.11. Similarly, it is indexed by a computation type, a pre- and a post-condition.

Precondition $\lambda(\mathsf{h}_0 : \ \mathsf{mem}) \rightarrow \top$ maps memories h_0 to the statement $\top$: `main` has no precondition. Postcondition $\lambda \mathsf{h}_0\ \ \mathsf{r}\ \ \mathsf{h}_1 \rightarrow \mathsf{r} \geq \texttt{0l}$ maps result r and the initial and final memories h_0-h_1 to the statement $\mathsf{r} \geq \texttt{0l}$: the outcome of `main` shall be greater or equal to zero. The proof obligation of function `main` is computed according to the weakest-precondition calculus the effect **`ST`** implements, as explained in Section 2.2.6.

In the example of Figure 4.1, the while loop has to be annotated with simple invariants, such as i being between 0 and 4, or l being live in memory. An abstract interpreter is well-suited to discover and infer such invariants statically. The memory model of the abstract interpreter we formalized in Chapter 3 is too weak, and cannot handle arrays, thus cannot either handle this *find* program. However, for instance Verasco [Jou+] can check our *find* program free of run-time errors by inferring these required

```
let main (): ST int32 (λh0 → ⊤) (λh0 r h1 → r ≥ 0l) =
    let i =   malloc root 0ul 1ul in
    let l = ❶ malloc root 0l 5ul in
    ❷ assign_list [-1l; -2l; -42l; -3l; -5l] l;
    let test_pre (h: mem): Type0
      = live h i     ∧ live h l
      ∧ length i = 1 ∧ length l = 5
    in
    let test_post (r: bool) (h: mem): Type0
      = live h i ∧ live h l
      ∧ r == (  get h i 0 < 5ul
             && get h l (v (get h i 0)) < 0l )
    in
    let test (): Stack bool test_pre (λ_ r h → test_post r h)
      = if !*i < 5ul then l.(!*i) < 0l else false
    in
    ❹ while test (λ() → i *= (!*i + 1ul));
    ❺ if !*i ≥ 5ul then failwith "Not found!" else l.(!*i)
```

Fig. 4.2: *find* program expressed in Low*.

```
let main (): ST_h ⊤# int32 (λh0 → ⊤) (λh0 r h1 → r ≥ 0l) =
    let i, l = malloc root 0ul 1ul, malloc root 0l 5ul in
    assign_list [-1l; -2l; -42l; -3l; -5l] l;
    let test () = if !*i < 5ul
                  then l.(!*i) < 0l else false in
    let body () = !*i ← !*i + 1ul in
    while test body;
    if !*i ≥ 5ul then failwith "Not found" else l.(!*i)
```

Fig. 4.3: The program *find* expressed in the (hypothetical) effect **ST**$_h$.

invariants. Solely resorting to abstract interpretation would of course not be satisfactory for our purpose: we would then lose the advantages brought by dependent types, that is, expressiveness.

Lighter Annotations Using an Enriched Effect Instead of choosing between abstract interpretation and dependent types, we propose to combine them. We aim at augmenting effects by abstract interpretation. Section 4.4 defines such a hybridization by considering a simple effect designed for the purpose of our demonstration. To elaborate further on the *find* example, consider **ST**$_h$, the hypothetical hybridization of the effect **ST**. Just as **ST**, **ST**$_h$ computes proof obligations. While computing proof obligations, **ST**$_h$ also performs static analysis that automatically generates additional invariants. These "properties for free" lighten the *hybrid* proof obligations. In **ST**$_h$, function `main` in Figure 4.3 would require no manual annotation.

4.2 Anatomy of our Weakest-Precondition Monad Transformer

Our aim is to enhance weakest-precondition monads with abstract interpretation techniques: for that purpose, we want to define a transformer of such monads, producing *hybrid* monads that embed abstract interpretation. This section focuses on describing which kinds of inputs this hybridization is fed with, and which kinds of outputs it produces.

There exists no unique shape for specification monads working with weakest-preconditions. Such monads can be formulated in various ways, and express a lot of different features. We therefore impose some restrictions on the monads we consider: Section 4.2.1 describes the various type-signatures a monad should conform to in order to be eligible for our transformation. Similarly, abstract interpretation is a very broad domain, and an abstract interpreter has no canonical form; Section 4.2.2 states what we assume in terms of semantics, soundness, type signatures and behavior.

4.2.1 Weakest-Precondition Monads

As stated in Section 2.2.4, a weakest-precondition monad is a monad whose representation is a weakest-precondition. In this chapter, a weakest-precondition shall be of the type (given below) `wp' st w τ`, for some `st`, `w` and τ. The type `wp' st w τ` is inhabited by weakest-preconditions about possibly stateful[1] computations producing values of type τ wrapped in some indexed type `w`. To encode partial computations for instance, one can let `w` be the indexed type `option`.

[1]One can always take `st` to be non-informative, e.g. `unit`.

```
type pre (st: Type) = st → prop
type post (st: Type) (w: Type → Type) (t: Type)
  = st → w t → prop
type wp' st w τ = post st w τ → pre st
```

A weakest-precondition `wp' st w τ` is defined as a map from post-conditions to pre-conditions. A pre-condition is a predicate about an initial state of type `st`, while a post-condition is a predicate about a final state and a wrapped value of type `w t`.

Remember in Section 2.2.2 that weakest-precondition transformers and Hoare logic are closely related. Consider the fragment of code `c`, and `w` its corresponding weakest-precondition (transformer). Given any `p` post-condition, by construction of `w`, if the pre-condition `w p` holds, then the post-condition `p` holds after the evaluation of `c`. In other words, for all post-conditions `p`, the Hoare triple `{w p}c{p}` holds. From this relation to Hoare triples, `w` should be a monotonic map: the pre-condition `w p` shall never get easier when `p` gets stronger. The type $\mathtt{wp}_{\mathtt{mon}}$ below avoids such inconsistencies by ensuring monotony via a refinement.

```
let post_order #st #w (p q: post st w τ)
  = ∀ (r: w τ) (s: st). p s r ⟹ q s r
let monotony #st #w (f: wp' st w τ)
  = ∀(p q: post st w τ). post_order p q ⟹ (∀s. f p s ⟹ f q s)
let wp_mon st w τ = f: wp' st w τ {monotony f}
```

Finally, we require our monads to provide the standard return and bind operations, along with an `if` operation. The record type $\texttt{monad}_{\texttt{wp}}$ recaps all the information related to what we expect a weakest-precondition monad to be. It might provide any number of other actions (i.e. a stateful *store* operation), whence the field `actions`.

```
type monad_wp = {
  st: Type;
  w: Type → Type;
  return: x:τ → wp_mon st w τ;
  bind: wp_mon st w τ → (τ → wp_mon st w β) → wp_mon st w β;
  if_: wp_mon st w ℤ → wp_mon st w τ → wp_mon st w τ → wp_mon st w τ;
  while: wp_mon st w ℤ → wp_mon st w unit → wp_mon st w unit;
  e_actions: ⋯;
}
```

4.2.2 Abstract Interpreter Interface

Under the term "abstract interpretation techniques", there exists a plethora of algorithms and theories. We presented in Chapter 3 an abstract interpreter. This section states the different assumptions we do over the abstract interpreters we consider. These assumptions are materialized by the record type `abint` below.

```
type abint = {
  M#: Type0;
  M: Type0;
  γ: M# → set M;
  order#: o: (M# → M# → bool) {antisym o ∧ transitive o};
  top: top:M#{∀ αs. αs `order#` top};
  widening: x:M# → y:M# → r:M# {order# x r ∧ order# y r};
  widening_lemma: ⋯;
  sequence#: (M# → M#) → (M# → M#) → M# → M#;

  exp: Type0 → Type u#1;
  γe: exp ℤm → M# → set ℤm;
  assume#: exp ℤm → M# → M#;

  ab_actions: ⋯;
}
```

An abstract interpreter that implements the type `abint` is equipped with abstract memories $\mathbb{M}^\#$ that concertize to concrete memories $\mathbb{M}$ via the function γ. Abstract states are ordered by $\texttt{order}^\#$ and `top` is the biggest abstraction for memories. A widening operator for abstract states shall be provided (fields `widening` and `widening_lemma`). An abstract interpretation of a program is an abstract state transformer, i.e. of type $\mathbb{M}^\# \to \mathbb{M}^\#$. The field `sequence`$^\#$ composes two such interpretations. The abstract interpreter shall also be capable of backward analysis for expression, whence the indexed type `exp` and the field `assume`$^\#$. The field γ_e computes the abstraction of the abstract semantics of an expression given an abstract memory. Finally, the field `ab_actions` has the same role as `e_actions` in the record $\texttt{monad}_{\texttt{wp}}$ above: the abstract interpreter is free to implement any number of other actions (for instance, the forward analysis of additions).

4.2.3 Typing our Transformer

Our monad and abstract interpreter transformer intuitively have the arrow type $\texttt{w}:\texttt{monad}_{\texttt{wp}} \to \texttt{ab}:\texttt{abint} \to \texttt{h}:\texttt{monad}_{\texttt{wp}}$, i.e. a map from weakest-precondition specification monads and abstract interpreters to weakest-precondition specification monads. However, we do not hybridize a monad with an abstract interpreter if they do not share the same semantics. Figure 4.4 illustrates such a connection. The weakest-precondition computation of some program `p` is demonstrated on the left, and its abstract interpretation on the right. The connection we are looking for is summarized by the "space of memories" illustration: we want the abstract interpretation to be a sound approximation of the semantics that the weakest-precondition yields. Following this illustration, a first condition is that the memory type `ab.`$\mathbb{M}$ (to which abstract memories `ab.`$\mathbb{M}^\#$ concretize to) should be equal to `w.st`. Then every action (`bind`, `sequence`...) implemented by `w` should have a semantically compatible counterpart as an action implemented by `ab`. We explore this relation more in-depth in Section 4.5.2.

Similarly, the output monad structure `h` should inherit some properties from `w`. We compute weakest-precondition to perform formal verification of programs: `h` should yield a sound weakest-precondition calculus suitable for verification as well. Such properties are discussed more in-depth in Section 4.5.1.

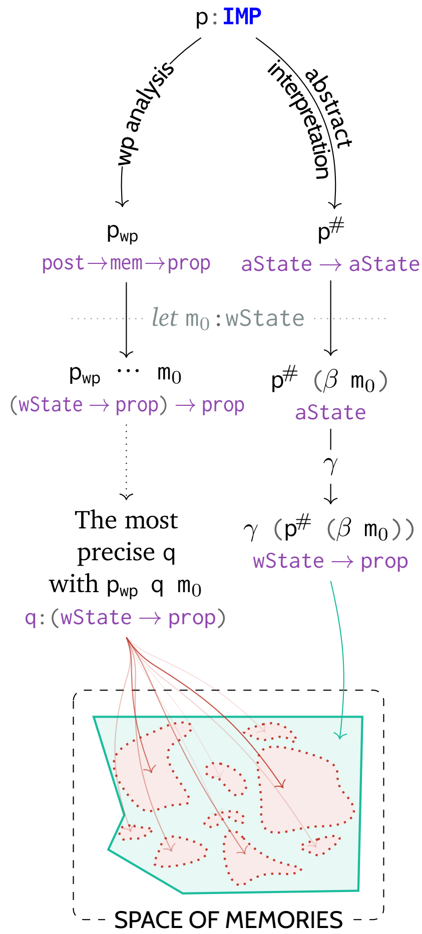

Fig. 4.4: Sound approximation of a weakest-precondition-defined semantics by an abstract semantics. On the left, the program `p` is given a precondition by a weakest-precondition calculus. On the right, `p` is interpreted abstractly. The weakest-precondition yields satisfiable preconditions only for certain states (the dotted shapes in red). The abstract state computed on the right is represented by the green continuous polygon. In the illustration, the abstract interpretation at stake over-approximates the semantics yielded by the considered weakest-precondition calculus: the memories in dotted red are indeed contained in the green continuous-line abstraction.

4.3 A Weakest-Precondition Monad and Abstract Interpreter for IMP^x

In order to present our idea of hybrid weakest-precondition monads, we instantiate our transformer with concrete inputs. This section defines an abstract interpreter $\texttt{W}^\#$`:abint` (Section 4.3.3) and a weakest-precondition monad `W:`**`effect`** (Section 4.3.2), that both target an imperative language similar to the one defined in Chapter 3 (Section 4.3.1).

The language IMP was designed to be simple enough so that Chapter 3 could present a full implementation of its abstract interpretation. Our aim is different here, and the memories IMP models are too simple to represent arrays, and thus are too weak to encode our example *find*. As a consequence, Section 4.3.1 defines another imperative language, IMP$^{\mathsf{x}}$. We conjecture the existence of a sound abstract interpreter written in F*, similar to the one presented in Chapter 3, but that analyses IMP$^{\mathsf{x}}$. This trio –IMP$^{\mathsf{x}}$, `W` and `W#`– is used throughout this chapter to define and illustrate our hybridization method.

4.3.1 Defining the Language IMP$^{\mathsf{x}}$

First, we need to define a simple language in which it is still easy to write imperative programs like our *find* example. To this end, we consider the IMP$^{\mathsf{x}}$ language presented in Eq. (4.1), that operates on memories of type $\mathbb{M}$, mapping variable names to fixed-length arrays of numbers. The initial memory maps every variable name to a zero-length array.

```
type varname = string
type binop = | Plus | Minus | Mult
             | Eq | Lt | And | Or
type expr: Type → Type =
  | Const: (#a:Type) → a → expr a
  | Deref: varname → expr ℤ → expr ℤ
  | BinOp: binop → expr ℤ → expr ℤ → expr ℤ
type stmt =
  | Alloc:  varname → ℤ → stmt
  | Assign: varname → expr ℤ → expr ℤ → stmt
  | Seq:    stmt → stmt → stmt
  | If:     expr ℤ → stmt → stmt → stmt
  | While:  expr ℤ → stmt → stmt
```
(4.1)

IMP$^{\mathsf{x}}$ has numeric expressions `expr` and instructions `stmt`; **`Const`** constructs constants, *false* is encoded as 0, *true* as any other number. Variable names are of type `varname`. All variables in IMP$^{\mathsf{x}}$ are mapped in memory to arrays, and we use arrays of size one to manipulate scalar variables. In the rest of this document, i and lst are two variable names of type `varname`. The expression **`Deref`** `v i` dereferences the `i`th item of the array at variable `v` in the store, as presented in rule (4.2).

$$\frac{M \vdash \texttt{i} \downarrow i' \quad M[v] = \langle a_0 \dots a_{i'} \dots \rangle}{M \vdash \texttt{Deref v i} \downarrow a_{i'}} \tag{4.2}$$

The instruction **`Alloc`** `v c` allocates a zero-filled array of size `c` to the variable `v` in the store (Rule (4.3)). If `i` is an expression that evaluates to i' and `e` evaluates to e', then **`Assign`** `v i e` stores the value `x'` at the `i'` offset of the array at the variable name `v` (Rule (4.4)).

$$\frac{\texttt{c is a positive constant}}{M \vdash \texttt{Alloc v c} \downarrow M\{\texttt{v} \mapsto \underbrace{\langle 0 \dots 0 \rangle}_{\text{length c}}\}} \tag{4.3}$$

$$\frac{M \vdash \texttt{i} \downarrow i' \quad M \vdash \texttt{e} \downarrow e' \quad M[v] = \langle a_0 \dots a_{i'} \dots \rangle}{M \vdash \texttt{Assign v i e} \downarrow M\{v \mapsto \langle a_0 \dots e' \dots \rangle\}} \tag{4.4}$$

Just as in Section 3.3 of Chapter 3, $\texttt{osem}_{\texttt{expr}}$ and $\texttt{osem}_{\texttt{stmt}}$ implement the semantics of expressions and statements of our language, partially given by the rules above.

```
let rec osem_stmt (m0: 𝕄) (s: stmt): set 𝕄 = ⋯
let rec osem_expr (m0: 𝕄) (e: expr τ): set τ = ⋯
```

They resemble the ones of Chapter 3, thus we omit their definition and give only their signature.

4.3.2 W: A Dijkstra Monad for IMP^x

The memory model underlying the language IMP^x is a map from variable names to arrays of integers. The language IMP^x does not feature exceptions or such other mechanisms. In consequence, the representation for our weakest-precondition monad is $\texttt{wp}'_{\texttt{w}}$, defined below. We also define type synonyms for pre- and post-conditions.

```
type pre_w      = pre   𝕄
type post_w  t  = post  𝕄 id t
type wp'_w   t  = wp'   𝕄 id t
type wp_w^mon t = wp_mon 𝕄 id t
```

The two monadic operations bind and return are uncomplicated and very similar to those defined previously in Section 2.2.4. Note that the weakest-precondition of an expression `Const x` is `return x`, for any `x`.

```
let bind_w (f: wp_w^mon τ) (g: id τ → wp_w^mon ν): wp_w^mon ν
  = λ(p: post_w ν) (s: st_w) → f (λ(s': st_w) v → g v p s') s
let return_w (v:τ): wp_w^mon τ = λp s → p s v
```

Following the definition of the language IMP^x, we now define actions that handle expressions. To this end, the function $\texttt{liftBinOp}_{\texttt{w}}$ takes a binary operation and lifts it as a binary action of our monad. The helper function `concrete_op_of_binop` maps a binary operation of type `binop` to an actual numerical binary operation.

```
let concrete_op_of_binop: binop → ℤ → ℤ → ℤ
  = λop → match op with
         | Plus → (λx y → x + y)
         | Lt   → (λx y → if x<y then 1 else 0)
         | …
let liftBinOp_w (op: binop): ℤ → ℤ → wp_w^mon ℤ
  = λ(x y: ℤ) → return_w (concrete_op_of_binop op x y)
```

Using this helper function, it is easy to define various actions that compute the weakest-preconditions of expressions involving binary operators. For example, below we define $\texttt{addition}_{\texttt{w}}$: if `x` and `y` are two integers, then $\texttt{addition}_{\texttt{w}}$ `x y` is the weakest-precondition of `BinOp Plus (Const x) (Const y)`. Given $\texttt{x}_{\texttt{e}}$ and $\texttt{y}_{\texttt{e}}$ two expressions, and $\texttt{x}_{\texttt{wp}}$ and $\texttt{y}_{\texttt{wp}}$ their weakest-preconditions, the weakest-precondition of **BinOp** **Plus** x_e y_e can be computed by binding x_{wp} and y_{wp} with the $\mathtt{bind}_w$ action we defined above: $\mathtt{bind}_w\ x_{wp}\ (\lambda x \to \mathtt{bind}_w\ y_{wp}\ (\mathtt{addition}_w\ x))$.

```
let addition_w : ℤ → ℤ → wp_w^mon ℤ = liftBinOp_w Plus
let eq_w       : ℤ → ℤ → wp_w^mon ℤ = liftBinOp_w Eq
let minus_w , lt_w , and_w , ⋯ = ⋯
```

After showing how to compute the weakest-preconditions of expressions involving binary operations and constants (with $\mathtt{return}_w$), the last expression operators to take care of are the dereference operator and the assignment statement. They both resemble the actions $\mathtt{assign}_w$ and $\mathtt{read}_w$ a lot, defined in Section 2.2.4.

Let v a variable name, i an index, p a post-condition and s_0 an initial state. The pre-condition $\mathtt{deref}_w\ v\ i\ p\ s_0$ for the expression **Deref** v (**Const** i) consists in proving the post-condition p given s_0 as final state and the dereferenced value `index` $(s_0\ v)$ i as outcome[2]. The pre-condition $\mathtt{assign}_w\ v\ i\ x\ p\ s_0$ for the instruction **Assign** v (**Const** i) (**Const** x) consists in proving p for the updated memory where the ith value pointed by variable v is x.

[2] With `index l i` being the function that looks up the ith value of the list l. The type of function `index` is the dependent arrow $\mathtt{l}:\mathtt{list}\,\tau \to \mathtt{i}:\mathbb{N}\{\mathtt{i}<\mathtt{length}\ \mathtt{l}\} \to \tau$.

```
let deref_w (v: varname) (i: ℤ): wp_w^mon ℤ
  = λ(p: post_w ℤ) (s: 𝕄) →
        i ≥ 0 ∧ i < length (s v)
      ∧ p s (index (s v) i)
let assign_w (v: varname) (i: ℤ) (x: ℤ): wp_w^mon unit
  = λ(p: post_w unit) (s: 𝕄) → i ≥ 0 ∧ i < length (s v)
          ∧ p (mem_upd s v (arr_upd (s v) i x)) ()
```

The weakest-precondition of the sequence of two instructions a and b is defined by the monadic composition of the weakest-preconditions of a and the constant function $\lambda\_\to b$. The combinator `if` is straightforward. The action that handles the instruction **While** requires a loop invariant `inv` as a parameter (it has no equivalent in IMP^x).

```
let sequence_w (f: wp_w^mon unit) (g: wp_w^mon unit): wp_w^mon unit
  = bind_w f (λ_ → g)
let if_w (c: ℤ) (a: wp_w^mon τ) (b: wp_w^mon τ): wp_w^mon τ
  = λp s → if c=0 then b p s else a p s
let while_w (inv: 𝕄 → prop) (c: wp_w^mon ℤ) (body: wp_w^mon unit)
  : wp_w^mon unit
  = λp s0 → inv s0
      ∧ (∀ (s1: 𝕄 {inv s1}).
           let f c': wp_w^mon unit
             = λq s2 → if c' = 0 then q s2 ()
                       else (inv s2 ∧ body (λ s3 _→inv s3) s2)
           in bind_w c f p s1 )
```

Consider the IMP^x loop **While** `cond lbody`, with `cond`$:$`expr`$\,\mathbb{Z}$ its condition and `lbody`$:$`stmt` its body. Let p be a post-condition of the loop. Let c be the weakest-precondition of the expression cond and body be the one of the statement lbody. $\texttt{while}_w$ inv c body is the weakest-precondition of the while loop w.r.t. the invariant inv. It requires inv to hold before the loop (inv s_0), and after any evaluation of lbody in a state satisfying the invariant (body $(\lambda s_3 \rightarrow \texttt{inv}\ s_3)\ s_2$). The post-condition p should hold for any state in which inv holds and cond evaluates to zero.

Example To illustrate how weakest-preconditions are computed, let us reconsider the example *find* of Figure 4.1 and focus on the first four lines. Here, the loop invariant is put on loop entry before the entry test, while the invariant presented in Figure 4.1 and in Section 4.1 was put after the entry test. By composing the definitions of this section, the following definition example computes the weakest-precondition of the example up to the While loop.

```
let ( >>= ) = bind_w     // Shortcuts for easier use
let ( >> ) = sequence_w  // of bind and sequence
let example i lst: wp_w^mon unit =
  let inv s: prop
    = length (s i) = 1 ∧ length (s lst) = 5
    ∧ ( let i = deref_spec s i 0 in
        i < 5 ∧ i ≥ 0 ∧ deref_spec s lst i ≥ 0 )
  in let condition: wp_w^mon ℤ =
        deref_w i 0 >>= (λi → lt_w i 5)
    >>= (λx → deref_w i 0 >>= deref_w lst
                          >>= lt_w 0 >>= and_w x)                (4.5)
  in let body: wp_w^mon unit
    = deref_w i 0 >>= addition_w 1 >>= assign_w i 0
  in alloc_w i 1 >> alloc_w lst 5
  >> assign_w lst 0 (-1) >> assign_w lst 1 (-2)
  >> assign_w lst 2  42  >> assign_w lst 3 (-3)
  >> assign_w lst 4 (-5)
  >> while_w inv condition body
```

Semantics. Let us review the relation between the weakest-precondition calculus we just defined and the semantics of IMP^{x}. For the sake of simplicity, we only consider the allocation of an array of size n at variable v. Below, alloc_sem_lemma connects $\texttt{alloc}_w$ with the operational semantics of the statement **Alloc** v n (let us denote it s).

```
let set_post (p: post_w unit): set 𝕄
  = λs → p s ()
let alloc_sem_lemma (v: varname) (n: ℤ) (m₀: 𝕄)
  (p: post_w unit {osem_stmt m₀ (Alloc v n) ⊆ set_post p})
  : Lemma ((∃ r. r ∈ osem_stmt m₀ (Alloc v n))
       ⟺ m₀ ∈ alloc_w v n p)
  = ()
```

We consider any initial memory $\mathtt{m}_0$. In Figure 4.5, the semantics of s (the arrow $\mathtt{osem}_{\mathtt{stmt}}\ \mathtt{m}_0\ \mathtt{s}$) connects this initial memory to three final memories, $\mathtt{m}_1$, $\mathtt{m}_2$ and $\mathtt{m}_3$. Then we consider any post-condition p that *includes*[3] the final memories characterized by the semantics of s from $\mathtt{m}_1$. In the figure, such a p is illustrated by $\mathtt{p}_1$, $\mathtt{p}_2$ and $\mathtt{p}_3$: the three of them indeed include $\mathtt{m}_1$, $\mathtt{m}_2$ and $\mathtt{m}_3$.

The expression $\mathtt{alloc}_{\mathtt{w}}$ v n p should be the weakest-precondition so that evaluating the statement **Alloc** v n yields the post-condition p. As pre-conditions are predicates over memories (that is sets of memories), let us rephrase: $\mathtt{alloc}_{\mathtt{w}}$ v n p is the set of admissible initial memories for the allocation to produce a final memory that belongs to the set p. In yet other terms, if $\mathtt{alloc}_{\mathtt{w}}$ v n p computes a correct weakest-precondition, then it should include the initial memory $\mathtt{m}_1$ since p *includes* the final memories $\mathtt{osem}_{\mathtt{stmt}}\ \mathtt{m}_0\ \mathtt{s}$.

The double implication comes from the fact an invalid allocation (e.g. **Alloc** i (-3)) yields an empty $\mathtt{osem}_{\mathtt{stmt}}\ \mathtt{m}_0\ \mathtt{s}$.

[3] A post-condition p of type $\mathtt{wp}_{\mathtt{w}}$ **unit** is a predicate about **unit** values and memories: in other terms, p is simply a predicate about memories, that is, a set. This is exactly what the function $\mathtt{set}_{\mathtt{post}}$ does.

4.3.3 $\mathtt{W}^\#$: Abstract Interpretation for IMP$^{\mathsf{x}}$

This section details our conjecture that an abstract interpreter written and proven sound in F⋆ is easy to write for our language IMP$^{\mathsf{x}}$, in a similar fashion to the one described in 3. Let us call this abstract interpreter $\mathtt{W}^\#$. It has abstract memories of type $\mathbb{M}^\#$, and enjoys abstract semantics $\mathtt{asem}_{\mathtt{expr}}$ and $\mathtt{asem}_{\mathtt{stmt}}$ for IMP$^{\mathsf{x}}$ expressions and statements with builtin soundness as type refinement.

```
val 𝕄#: Type
instance aState_adom: adom 𝕄# = { c = 𝕄 ; ⋯ }
val asem_expr (m#_0: 𝕄#) (e: expr ℤ)
  : (r:itv{∀(m_0:𝕄). m_0 ∈ γ m#_0 ⟹ osem_expr m_0 e ⊆ γ_e r})
val asem_stmt (s: stmt) (m#_0: 𝕄#)
  : (m#_1: 𝕄# {∀ (m m': 𝕄). (m  ∈ γ m#_0 ∧ m' ∈ osem_stmt m s)
                              ⟹ m' ∈ γ m#_1 })
val assume#: 𝕄# → expr ℤ → 𝕄#
```

Below we define the actual definition of $\mathtt{W}^\#$, in the form of a record of type abint, as we described in Section 4.2.2.

```
let W# = { 𝕄#  = 𝕄#; 𝕄 = st_w; γ = aState_adom.γ
; γ_e = (λe s# → itv_γ (asem_expr s# e))
; order# = aState_adom.corder  ; top = aState_adom.top
; widening = aState_adom.widen; widening_lemma = ⋯
; sequence# = (λf g s# → g (f s#)); exp = expr
; assume# = assume#; ab_actions = ⋯ }
```

The pair ($\mathtt{W}^\#$, W) will now serve as an example input for the effect transformer we try to define here. After focusing on the input, in the next section (Section 4.4), we will focus on the output of our transformer.

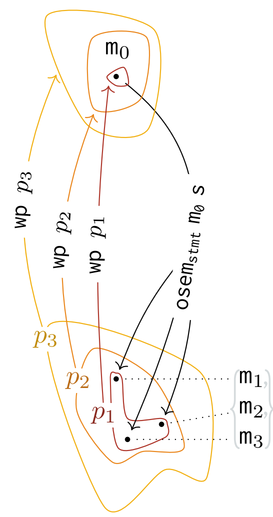

Fig. 4.5: Connection between the operation semantics of a statement s with its corresponding weakest-precondition wp.

4.4 $\mathtt{W}^h$: Hybridization of W and $\mathtt{W}^\#$

This section illustrates our transformation by the definition of the weakest-precondition monad $\mathtt{W}^h$ that arranges together $\mathtt{W}^\#$ and W. In this aim, Sections 4.3.2 and 4.3.3 instantiate *find* on $\mathtt{W}^\#$ and W to produce the invariant inv of (4.5): the index i refers to an array of size one in memory, while the list is of size five and the $\mathtt{i} \in [0,4]$. This invariant is the one required by the weakest-preconditions of W for the loop.

4.4.1 Hybrid State, Values and Weakest-Preconditions

The hybridization combines the abstract and concrete views of the same program. Structurally, $\mathtt{W}^h$ acts as a cartesian product of our monad W and our abstract interpreter $\mathtt{W}^\#$. A computation f is seen as hybrid: the abstract and W monadic representation of f are superposed. As a result, the representation of computations producing τ values in $\mathtt{W}^h$ is a weakest-precondition on hybrid values hVal τ, dealing with hybrid states $\mathbb{M}_h$. Both types are defined below, as well as types for hybrid postconditions, preconditions and weakest-preconditions. The first and second items of a tuple are returned by fst and snd.

```
type 𝕄_h = 𝕄 × 𝕄#              type wp_h τ = post_h τ → pre_h
type hVal τ = τ × expr τ        type post_h τ = 𝕄_h → hVal τ → prop
type pre_h = 𝕄_h → prop         val fst: τ×β → τ      val snd: τ×β → β
```

4.4.2 Actions Computing Weakest-Precondition for Expressions

We now define the actions that allow our $\mathtt{W}^h$ monad to compute weakest-preconditions of the expressions that IMP$^\mathsf{x}$ allows.

First, let us focus on the expressions that consist in a numerical binary operation. Such expressions are of the shape **BinOp** op $\mathtt{x_e}$ $\mathtt{y_e}$, where op is a numerical binary operation (e.g. **Plus** or **Eq**), and where $\mathtt{x_e}$ and $\mathtt{y_e}$ are both expressions. The helper function $\mathtt{liftBinOp_w}$ of Section 4.3.2 lifted binary operations to the monad W. The function $\mathtt{liftBinOp_h}$ is the hybridization of $\mathtt{liftBinOp_w}$: given a binary operation, it builds up on $\mathtt{liftBinOp_w}$ to interleave some abstract bits. It takes a binary operation of type binop as first parameter, and returns a hybrid action indexed by two hybrid integers.

The abstract expression bits of these two hybrid integers are of type $\mathtt{W}^\#$.exp, which is defined as expr. The hybrid integers are tuples: those are destructed at ❶ as the tuples $(\mathtt{x},\ \mathtt{x}^\#)$ and $(\mathtt{y},\ \mathtt{y}^\#)$. Computing the binary operation op on the abstract expressions $\mathtt{x}^\#$ and $\mathtt{y}^\#$ in our case trivially amounts to the expression **BinOp** op $\mathtt{x}^\#$ $\mathtt{y}^\#$, denoted $\mathtt{r}^\#$ at ❷. At ❸, we return a weakest-precondition of type $\mathtt{wp_h}\ \mathbb{Z}$. Given a post-condition p and a hybrid memory $(\mathtt{s},\ \mathtt{s}^\#)$ –s a concrete memory and $\mathtt{s}^\#$ an abstract one– we return a precondition based on $\mathtt{liftBinOp_w}$ op, that is, the action of type $\mathbb{Z} \to \mathbb{Z} \to \mathtt{wp_w}\ \mathbb{Z}$ we are seeking to hybridize. The concrete bits x, y and s are fed to $\texttt{liftBinOp}_w$ op, which is an action that expects concrete values, not hybrid ones. However, the weakest-precondition we define at ❸ receives a hybrid post-condition p of type $\texttt{post}_h$ $\mathbb{Z}$. The post-condition at ❹ adds a free hypothesis[4] (❺) from the abstract interpreter: the concrete result r is in the approximation $r^{\#}$. Then, the non-hybrid post-condition ❹ hands over to the hybrid one p, by crafting tuples: the new hybrid state consists in the new concrete state s' with the unchanged abstract state $s^{\#}$, while the hybrid value consists in the concrete value r (passed to the post-condition ❹ by $\texttt{liftBinOp}_w$ op) with the expression $r^{\#}$.

```
let liftBinOph (op: binop): hVal ℤ → hVal ℤ → wph ℤ
  = ❶ λ(x, x#) (y, y#) →
      let ❷ r# = BinOp op x# y# in
      ❸ λp (s, s#) →
        liftBinOpw op x y (❹ λs' r →
          ❺ r ∈ W#.γv r# s# ⟹ p (s',s#) (r,r#)
        ) s
```

Similarly to Section 4.3.2, having defined the binary operation in the form of the indexed hybrid action $\texttt{liftBinOp}_h$, it becomes trivial (see below) to write down the actions $\texttt{addition}_h$, $\texttt{subtraction}_h$, $\texttt{eq}_h$, etc.

```
let additionh = liftBinOph Plus
let subtractionh = liftBinOph Minus
let eqh = liftBinOph Eq
```

Dereference operator, allocation and assignement are very similar to what is done in $\texttt{liftBinOp}_h$.

[4]Such an hypothesis consists in running a forward analysis of the expression $r^{\#}$ in the abstract memory $s^{\#}$. A simple –and thus not very powerful– example of such a hypothesis with the abstract domain of intervals is yielded by the call $\texttt{liftBinOp}_h$ `Plus (a, Var A) (b, Var B) p (s, s#)` with $s^{\#}$ an abstract memory such that “a∈γ s[A]”, “b ∈ γ s[B]” and “s[A]=s[B]=[0,5]”, and with p a post-condition. In this case, ❺ amounts to r ∈ [0,10]: proving the post-condition p is made easier using this fact. Obviously, for such a trivial example, the benefit is nonexistent: an SMT solver is of course able to figure such an invariant automatically.

```
let derefₕ (vn: varname): hVal ℤ → wpₕ unit
  = λ(i, i#) →
      let r# = Deref vn (Const i#) in
      λp (s, s#) →
        deref_w vn i (λs' r →
          s' ∈ W#.γᵥ r# s# ⟹ p (s', s#) (r, r#)
        ) s

let allocₕ (vn: varname) (n: ℤ): wpₕ unit
  = λp (s₀, s#₀) →
      let s#₁ = asem_stmt (Alloc vn n) s#₀ in
      var_w vn (λs₁ () →
        s₁ ∈ W#.γ s#₁ ⟹ p (s₁, s#₁) ((), ()#)
      ) s

let assignₕ (vn: varname): hVal ℤ → hVal ℤ → wpₕ unit
  = λ(i, i#) (x, x#) →
      λp (s, s#₀) →
        let s#₁ = asem_stmt (Assign vn i# x#) s#₀ in
        assign_w vn i x (λs' () →
          s' ∈ W#.γ s#₁ ⟹ p (s', s#₁) ((), ()#)
        ) s
```

Now, let us introduce some rationale and intuition behind hybrid weakest-preconditions: we explain the link between hybrid weakest-preconditions and regular proof obligations. Consider $\mathtt{addition}_\mathtt{h}$ `x y` (given some `x` and `y`): the resulting predicate maps a postcondition and a hybrid state (that is, both a concrete and abstract state) to a proof obligation relative to the concrete state, lightened by assumptions brought by the abstract state. While this doesn't feel like a weakest-precondition at first glance, such hybrid weakest-preconditions can however be *reified* as regular weakest-preconditions. Given an abstract memory $s^\#$ and a concrete memory `s` abstracted by $s^\#$, the abstract knowledge accumulated from $s^\#$ can be used to lighten the proof obligations.

```
λ(p:post_w ℤ) (s:𝕄).
  s ∈ W#.γ s# ⟹
    addition_w x y (λs' (r, r#) → x s' r)
                   (s, s#)
```

4.4.3 Hybrid Monadic Operators

To define the hybrid Dijkstra monad $\mathtt{W}^h$, we need a bind and a return operator. In the case of `W`, these operations are very canonical. However, a *return* and a *bind* operation can implement very diverse behaviors, handle errors for instance. We choose the functions $\mathtt{return}_\mathtt{h}$ and $\mathtt{bind}_\mathtt{h}$, to directly inherit from $\mathtt{return}_\mathtt{w}$ and $\mathtt{bind}_\mathtt{w}$.

```
let returnₕ (v: τ): wpₕ τ =
  λp (s, s#) → return_w v (λs' v' → p (s', s#) (v', Const v)) s
```

Given a value v, $\texttt{return}_\texttt{h}$ crafts the value `(v', Const v)` (with `v'` being morally `v`). This hybrid value is fed to the given postcondition `p`, inside a lambda abstraction given as postcondition to $\texttt{return}_\texttt{w}$. The bind operator is more interesting: a composition of given hybrid computations `f` and `g` is a particular composition of the regular views of these computations. Recall the cartesian product nature of our hybridization (Section 4.4.1): structurally, a hybrid postcondition is a postcondition in `W` that carries an abstract view of memory and values. The same holds for hybrid weakest-preconditions. $\texttt{post}_h^{\downarrow}$ reformulates a hybrid postcondition $\texttt{p: post}_\texttt{h}\ \tau$ as a postcondition of type $\texttt{post}_\texttt{w}\ (\texttt{hVal}\ \tau \times \mathbb{M}^\#)$.

```
let post↓ₕ (p: postₕ τ): post_w (hVal τ×𝕄#)
  = λs (v, a) → p (s, a) v
let post↑ₕ (p: post_w (hVal ℤ×𝕄#)): postₕ ℤ
  = λ(s, a) v → p s (v, a)
let wp↓ₕ (w: wpₕ τ) (a: 𝕄#): wp_w (hVal τ × 𝕄#)
  = λ(p: post_w (hVal τ × 𝕄#)) (s: 𝕄) → w (post↑ₕ p) (s, a)
let wp↑ₕ (w: wp_w (hVal ℤ × 𝕄#)): wpₕ ℤ
  = λp s → w (post↓ₕ p) (fst s)
let bindₕ (f: wpₕ τ) (g: hVal τ → wpₕ τ): wpₕ τ
  = λ(p: postₕ τ) (s: 𝕄ₕ) →
     wp↑ₕ (wp↓ₕ f (snd s) `bind_w` (λ(v, a) → wp↓ₕ (g v) a)) p s
```

The hybrid bind down-lifts its hybrid arguments `f` and `g`, passes them to $\texttt{bind}_\texttt{w}$, and up-lifts the result to obtain a hybrid weakest-precondition: `g` is lifted in the lambda-abstraction right of $\texttt{bind}_\texttt{w}$ and given a value of type $\texttt{hVal}\,\tau \times \mathbb{M}^\#$ that injects the hybrid value in `g`, and lifts the result given the abstract state.

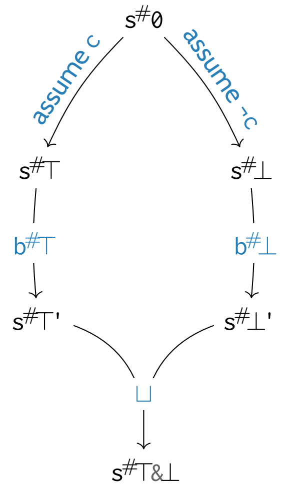

Fig. 4.6: Illustration of the abstract interpretation of a conditional statement whose condition is c and branches are $b^\#_\top$ and $b^\#_\bot$.

4.4.4 Conditional

A conditional alters the control-flow upon a condition. The abstract interpretation of a conditional is summarized by Figure 4.6: for each conditional branch, an abstract interpretation is run with an alteration to the initial abstract memory $s^\#_\emptyset$. This alteration consists in supposing that the condition is either true or false (the two uppermost arrows on Figure 4.6). The abstract analysis of the two branches results in two memories $s^{\#}_{\top}{}'$ and $s^{\#}_{\bot}{}'$, that are then joined together to form $s^\#_{\top\&\bot}$. This abstract memory $s^\#_{\top\&\bot}$ captures the abstract semantics of the conditional: whatever path taken, a concrete execution of the conditional is approximated by $s^\#_{\top\&\bot}$.

This last joining step is the culprit of the limitations our approach suffers from. Indeed, we encode abstract interpretations right into our hybrid weakest-precondition. This means that our hybridization manipulates abstract interpretations *through* weakest-preconditions reductions. Consider a computation `f`, its abstract interpretation $\texttt{f}^\#: \mathbb{M}^\# \to \mathbb{M}^\#$ and its regular and hybrid weakest-preconditions $\mathsf{f_{wp}} : \mathsf{wp_w}\ \tau$ and $\mathsf{f_{wp}}' : \mathsf{wp_h}\ \tau$. From $\mathsf{f_{wp}}'$, it is possible to extract a view of $\mathsf{f}^{\#}$ (see the *meta post-condition* $\mathsf{constant_{mp}}$ given in Section 4.6.1.1), but doing this inevitably generates some pre-conditions with respect to $\mathsf{f_{wp}}$. As a consequence, the two hybrid memories of the figure, $\mathsf{s}^{\#}_{\top}{}'$ and $\mathsf{s}^{\#}_{\bot}{}'$ cannot be computed in a same continuation, and thus $\mathsf{s}^{\#}_{\top\&\bot}$ is impossible to craft. For the sake of the presentation however, the definition of **if**$_\mathsf{h}$ supposes, for now, the existence of a joined $\mathsf{s}^{\#}_{\top\&\bot}$ at ❶.

```
let ifₕ : wpₕ ℤ → wpₕ unit → wpₕ unit → wpₕ unit
  = λ ❷ (c, c#) a b →
      λp (s, s#) →
        let assume_wp (cond: exp ℤ) (wp: wpₕ unit): w_w unit
          = λ(q: post_w 𝕄#) (s: 𝕄) →
              ❸ s ∈ assume# cond s# ⟹
                   wp (λ(s', ❹ _) _ → q s' ((), ()#)) s
        in
        let s#_⊤&⊥ = ❶ ⋯ in
        if_w c (❺ assume_wp        c#  a)
               (❻ assume_wp  (Not' c#) b)
               (λs' () → ❼ p (s',s#_⊤&⊥) ((), ()#))
               s
```

At ❷, the function **if**$_\mathsf{h}$ takes a hybrid condition and a weakest-precondition for each branch, a and b. The helper function **assume**$_\mathsf{wp}$ assumes (at ❸) some expression holds in the initial abstract memory $\mathsf{s}^{\#}$, and lifts a hybrid weakest-precondition wp as a regular one, dropping (at ❹) the computed abstract memory. This helper function is used at ❺ and ❻ to call **if**$_\mathsf{w}$. Finally, at ❼, we craft a regular post-condition out of p the hybrid post-condition to satisfy. This post-condition uses $\mathsf{s}^{\#}_{\top\&\bot}$.

4.4.5 While

Computing a weakest-precondition for a loop is achieved relative to a suitable invariant. By contrast, abstract interpretation infers such an invariant. Let us focus on this inference process. Consider $\mathsf{b} : \mathbb{M}^{\#} \to \mathbb{M}^{\#}$, an abstract loop body, and an initial abstract state $\mathsf{s}^{\#}_{0}$. The following sequence defines the cumulative abstract states for any n, $\mathsf{s}^{\#}_{n} \sqsubseteq^{\#} \mathsf{s}^{\#}_{n+1}$.

$$\mathsf{s}^{\#}_{n+1} \stackrel{\text{def}}{=} \mathsf{s}^{\#}_{n}\ \nabla\ \mathsf{b}\ \mathsf{s}^{\#}_{n}$$

The widening operator ∇ computes upper bounds in the state lattice and ensures the convergence of its iteration to reach a fixpoint, i.e., an m such that $s^{\#}_{m} = s^{\#}_{m+1}$. The abstract memory $s^{\#}_{m}$ is exactly a loop invariant for the head of the loop: a concrete state respects the invariant iff it is abstracted by $s^{\#}_{m}$. This construction can be mirrored in our hybrid weakest-precondition calculus. However, in such a setting, a body b is not an abstract state transformer anymore: it is a weakest-precondition, that is, a continuation. Then, a fixpoint for b, in terms of state, is computed as below, where s_{n+1} is equal to s_n, the fixpoint. This fixpoint is then passed to a continuation function `ct`.

$$\mathsf{b}\ (\lambda \mathsf{s}_1 \to \mathsf{b}\ (\lambda \mathsf{s}_2 \to \cdots\ \mathsf{b}\ (\lambda \mathsf{s}_{n+1} \to \mathsf{ct}\ \mathsf{s}_n)\ \mathsf{s}_n\ \cdots\ )\ \mathsf{s}_1)\ \mathsf{s}_0$$

Next, function `fp` computes an abstract state fixpoint in a similar way. It takes a weakest-precondition `f`, a continuation `ct`, concrete and abstract states `s` and $a^{\#}$`s`, and a fuel `n`. `fp` evaluates `f` with the postcondition ❶ that tests whether the abstract state is stable or not, using the widening operator `W#.widening`. If a fixpoint is reached (i.e. if condition at ❷ is true), it is passed to the continuation `ct v` $\mathsf{s}_0^{\#}$. Otherwise, we recurse with our new and widened abstract state $\mathsf{s}_2^{\#}$ (❸).

```
let rec fp (f:wpₕ τ) (ct:τ → 𝕄# → prop) (s:𝕄) (s₀#:𝕄#) (n:ℕ)
  : prop
  = if n=0 then ct v ⊤#𝕄 else
      f (❶ λ(s₁#, _) (v, _) →
             let s₂# = s₀# `W#.widening` s₁# in
             if ❷ s₂# = s₀# then ct v s₀#
                          else ❸ fp f ct s s₂# (n-1)
        ) (s, s₀#)
```

Since abstract interpretation and weakest-precondition computations are coupled in our hybrid setting, `fp` also necessarily computes a weakest-precondition given a concrete state. Note that each recursive call is performed with the same concrete state: we compute the same weakest-precondition, but with different abstract states, until stabilization. Having the function `fp` that computes abstract invariants, it is easier to define how hybrid `while` loops work.

Here, we are interested in computing the hybrid weakest-preconditions of statements of the shape **`While`** `c body`, with `c` an expression of type `expr` $\mathbb{Z}$ and `body : stmt` a statement. Let c_{wp} and $\mathsf{body}_{\mathsf{wp}}$ their respective hybrid weakest-preconditions, of type $\mathsf{wp}_{\mathsf{h}}\ \mathbb{Z}$ and wp_{h} **`unit`**. In these conditions, $\mathsf{while}_{\mathsf{h}}$ `inv` c_{wp} `body` is the hybrid weakest-precondition for **`While`** `c body`, given an invariant `inv` about concrete memories.

```
let whileh (inv: 𝕄 → prop) (cwp : wph ℤ) (bodywp : wph unit)
 : wph unit
 = let ❶ loopbody: wph (expr ℤ)
     = bindh cwp (λ(c, c#) →
           bindh (ifh (c, c#) bodywp (returnh ((), ()#)))
                 (λ_ → ❷ returnh c#))
   in
   λq (s, s#) →
   ❸ fp loopbody (❹ λc# s_i# →
   ❺ whilew (λs' → ❻ inv s' ∧ s' ∈ γ s_i#)
            (λp s' → ❼ cwp (λ(s, _) (c, _) → p s c) (s', s_i#))
            (λp s' → ❽ bodywp (λ(s, _) _ → p s ()) (s', s_i#))
            (λs' _ → ❾ q (s', assume# (!# c#) s_i#) ((),()#))
            s
    ) s s# 10
```

The hybrid computation that we conduct here takes place in two phases: first, finding a fixpoint in terms of abstract memories, second, handing over to the `W` action $\texttt{while}_{\texttt{w}}$.

First, at ❶, we let $\texttt{loop}_{\texttt{body}}$ be the weakest-precondition of the statement `If c body` *nothing*: just as in Chapter 3, since we look for a fixpoint, considering infinite loops instead of while loops is more convenient. The binding $\texttt{loop}_{\texttt{body}}$ is of type $\texttt{wp}_{\texttt{h}}$ `(expr` $\mathbb{Z}$`)`: it captures the expression $\texttt{c}^{\#}$ reflecting the conditional, which was obtained after binding $\texttt{c}_{\texttt{wp}}$. The function `fp` then (at ❸) computes a fixpoint for $\texttt{loop}_{\texttt{body}}$ in the form of $\texttt{s}_i^{\#}$, an abstract memory. The expression $\texttt{c}^{\#}$ of the loop condition and the invariant (abstract memory) $\texttt{s}_i^{\#}$ are provided by the continuation passed to `fp` at ❹.

Second, at ❺ we re-use the `W` action $\texttt{while}_{\texttt{w}}$. We craft a new invariant enriched with abstract interpretation at ❻, using the invariant computed by `fp`: the abstract memory $\texttt{s}_i^{\#}$. The weakest-preconditions $\texttt{c}_{\texttt{wp}}$ and $\texttt{body}_{\texttt{wp}}$ are transformed into non-hybrid weakest-preconditions (at ❼ and ❽) and then fed to $\texttt{while}_{\texttt{w}}$. Finally, ❾ is the post-condition supplied to $\texttt{while}_{\texttt{w}}$: we adapt the hybrid weakest-precondition `q` injecting $\texttt{s}_i^{\#}$', the abstract invariant assuming the loop condition does not hold anymore.

4.4.6 Functions and Reification

Language IMP neither defines procedures nor functions. We explained what hybrid weakest-preconditions are made of, but not how one can use them: this is what this subsection is intended for. Consider a program that sorts an array in memory: the specification "the program results in the list being indeed sorted" has nothing to do with abstract states or values. Abstract bits are not to be exposed to the user: they solely exist for inference purposes. The specification of such a "sort" example is tied to the behavior of "sort", and has nothing to do with our choice of analyzing our program through a regular or a hybrid weakest-precondition. Consider a function `f` mapping integers to integers of effect $\texttt{W}^h$, this section explains how to compute its weakest-precondition $\texttt{f}_{\texttt{h}}$, of type `hVal` $\mathbb{Z} \to \texttt{wp}_{\texttt{h}}$ $\mathbb{Z}$. $\texttt{f}_{\texttt{W}}$

is the reification of $\mathtt{f_h}$: a regular weakest-precondition, of type $\mathtt{wp_w}\ \mathbb{Z}$. By nature, $\mathtt{f_W}$ (i) approximates its input `x` by $\top^\#$, the expression holding no knowledge; (ii) approximates its initial concrete state by $\top^\#_{\mathbb{M}}$; (iii) injects the regular post-condition `p` in $\mathtt{f_h}$ by ignoring hybrid parts.

```
let f_W (x: ℤ): wp_w ℤ = λ(p: post_w ℤ) (s: 𝕄) →
    f_h (⊤#, x) (λ(_, s) (_, r) → p s r) (⊤#_𝕄, s)
```

However, approximating every concrete piece of data by the greatest element of the corresponding abstract lattice is weak. Commonly, an effectful Dijkstra monad is given a Hoare-style interface for writing specification: this is the case of Low⋆. For the sake of simplicity, we do not define such an interface here. Nonetheless, for a function whose input and initial state are constrained by a given precondition, it is possible to craft abstract expressions and states that approximate this precondition precisely, e.g., the abstract state $\{A \mapsto [11; \infty]\}$ from the precondition "variable A in memory is greater than 10".

In this section, we demonstrated how to embed and benefit from the abstract interpreter $\mathtt{W}^\#$ in our Dijkstra monad `W`. We now study the soundness of this calculus.

4.5 Statement of Soundness

Specification monads aim at program verification by computing proof obligations for programs. Section 4.5.1 details the generation of a proof obligation for an $\mathrm{IMP}^{\times}$ program with either a regular or a hybrid specification monad. Section 4.5.3 states a theorem of soundness.

4.5.1 Proof Obligations

A proof obligation is a formula to be proven, in order to ensure that a given program matches a given specification, consisting of pre- and post-conditions.

4.5.1.1 Regular weakest-preconditions

Recall the instructions `stmt` and expressions `expr` of the $\mathrm{IMP}^{\times}$ language (Section 4.3.1). Proving the specification $(pre, post)$ of a program `prg`: `stmt` correct using a weakest-precondition amounts to (i) computing `prg`'s weakest-precondition W, (ii) deriving a proof obligation from W, and (iii) proving that proof obligation. Below, $\mathtt{wp_w^{stmt}}$ and $\mathtt{wp_w^{expr}}$ give instructions or expressions a regular weakest-precondition by induction.

```
let rec wp_w^expr (e: expr ℤ): wp_w^mon ℤ
  = match e with
  | Const x      → return_w x
  | Deref v i    → bind_w   (wp_w^expr i) (deref_w v)
  | BinOp op a b → bind_w^bin (wp_w^expr a)
                             (wp_w^expr b) (liftBinOp_w op)

let rec wp_w^stmt (i: stmt): wp_w^mon unit
  = match i with
  | Alloc v n → alloc_w v n
  | Assign v i_e x_e →
      bind_w (wp_w^expr i_e)
             (λi → bind_w (wp_w^expr x_e) (assign_w v i))
  | Seq i j → sequence_w (wp_w^stmt i) (wp_w^stmt j)
  | If c_e t_b f_b → bind_w (wp_w^expr c_e)
                        (λc → if_w c (wp_w^stmt t_b)
                                     (wp_w^stmt f_b))
  | While inv e body → while_w inv (wp_w^expr e)
                                   (wp_w^stmt body)
```

A proof obligation is a statement to prove true so that a *specification* holds. Below PO_W `i pre post` computes the proof obligation that should hold to ensure that the program `i` respects its specification `(pre, post)`. The weakest-precondition for the evaluation of `i` starting at state s_W to satisfy the post-condition post is $(\mathrm{wp}_w^{\mathrm{stmt}}$ `i) post` s_W. This weakest-precondition should hold whenever the state s_W respects the precondition `pre`.

```
let PO_W (i: stmt') (pre: pre_w) (post: post_w unit)
  = ∀ (s_W: 𝕄). pre s_W ⟹ wp_w^stmt i post s_W
```

4.5.1.2 Hybrid weakest-preconditions

Similarly, $\mathrm{wp}_h^{\mathrm{stmt}}$ and $\mathrm{wp}_h^{\mathrm{expr}}$ map instructions and expressions to hybrid weakest-preconditions, the only differences being the nature (hybrid or regular) of the combinators.

```
let rec wp_h^expr (e: expr ℤ): wp_h^mon ℤ
  = match e with
  | Const x      → return_h x
  | Deref v i    → bind_h   (wp_h^expr i) (deref_h v)
  | ...

let rec wp_h^stmt (i: stmt): wp_h^mon unit                    (4.6)
  = match i with
  | Alloc v n → alloc_h v n
  | Assign v i_e x_e →
      bind_h (wp_h_of_exp i_e)
             (λi → bind_h (wp_h_of_exp x_e) (assign_h v i))
  | ...
```

The second step is to define how hybrid weakest-preconditions are translated into proof obligations. Whether we are dealing with hybrid or regular weakest-preconditions has no impact on the specification one is interested in. Thus, even to formulate a proof obligation using hybrid weakest-precondition, the specification is still described by regular pre- and post-conditions. Function PO_h hence defines a proof obligation given a hybrid weakest-precondition against a regular one.

```
type approx = pre:pre_w → s#: M# {∀ s. pre s ⟹ s ∈ γ s#}
let PO_h (α: approx) (f: instr') (pre: pre_w) (post: post_w unit)
 =∀(s_W:M). pre s_W⟹wp_h^stmt f (λ(s'_W,_) (r_W,_) → post s'_W r_W) (α pre,s_W)
```

In this proof obligation, the hybrid weakest-precondition is given a regular postcondition `post`, and applied on a hybrid state `(α pre, s_W)`. Given a precondition `pre`, `α:approx` constructs an abstract state that approximates `pre`: any concrete state that satisfies `pre` is approximated by `(α pre)`, that is $\mathtt{pre} \subseteq \mathtt{W}^{\#}.\gamma\ (\alpha\ \mathtt{pre})$.

4.5.2 Abstract Interpreter Soundness

The soundness proof of our hybrid weakest-precondition calculus relies on the fact the abstract interpreter `W#` is a sound abstract interpretation w.r.t. the semantics implemented by the regular weakest-precondition calculus `W`. This fact (the dashed arrow on Figure 4.7) can be derived from (i) the soundness of `W#` w.r.t. the operational semantics of IMP$^{\times}$ (arrow ❶ on the figure) provided by the refined type of $\mathtt{asem}_{\mathtt{stmt}}$ (Section 4.3.3), and (ii) the connection between `W` and the operational semantics of IMP$^{\times}$ (arrow ❷ on the figure), discussed in Section 4.3.2.

As an example, below `alloc_W#_W` is the **`Alloc`** instance for the dashed arrow on Figure 4.7. Given any variable name `v`, length `n`, abstract state $\mathtt{m}_0^{\#}$ that approximates an initial state $\mathtt{m}_0$, and `p` a post-condition, `alloc_W#_W v n` $\mathtt{m}_0$ $\mathtt{m}_0^{\#}$ `p` states that the regular weakest-precondition rules $\mathtt{alloc}_{\mathtt{w}}$ and the abstract semantics $\mathtt{asem}_{\mathtt{stmt}}$ `(Alloc v n)` reflects a same semantics.

```
let alloc_W#_W (v: varname) (n: ℤ)
  (m0: M) (m0#: M# {m0 ∈ γ m0#}) (p: post_w unit)
  : Lemma (alloc_w v n p m0 ⟹
           alloc_w v n (λm1 _ → m1 ∈ γ (asem_stmt (Alloc v n) m0#)
                                  ∧ p m1 ()
                       ) m0
          )
  = ⋯
```

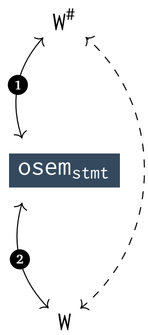

Fig. 4.7: Consistency of the semantics implemented by the regular weakest-precondition calculus `W`, the operational semantics $\mathtt{osem}_{\mathtt{stmt}}$, and the abstract semantics of `W#`. Each arrow is a relation between the semantics implemented by two entities. The dashed arrow is the relation we are looking for.

Definition 1 reformulates this notion, which is then generalized for any statement of IMP$^{\times}$ in Theorem 1.

Definition 1 (W# is sound w.r.t. W) *An abstract interpretation* $\mathtt{f}^{\#}$*, of type* $\mathbb{M}^{\#} \to \mathbb{M}^{\#}$*, is sound w.r.t. its* `W` `f`*, of type* $\mathtt{wp}_{\mathtt{w}}$ **`unit`***, if* `αsound f# f` *holds: given any initial abstract state* $\mathtt{s}_{\#}$ *and concrete state* $\mathtt{s}_{\mathtt{W}} \in \gamma\ \mathtt{s}_{\#}$*, proving a* *post-condition on* `f` *is stronger than proving that same post-condition and computed concrete state approximated by* γ_{ST} $(f^{\#}\ s_{\#})$.

$$
\begin{aligned}
&\mathbf{let}\ \alpha\mathsf{sound}\ (f^{\#}{:}\,\mathbb{M}^{\#} \to \mathbb{M}^{\#})\ (f{:}\ \mathbf{W}.\mathsf{wp}\ \mathbf{unit})\\
&\quad = \forall\ (p{:}\ \mathsf{post}_{\mathsf{w}}\ \mathbf{unit})\ (s^{\#}{:}\ \mathbb{M}^{\#})\ (s{:}\ \mathbb{M}).\ s \in \gamma_{ST}\ s^{\#} \Longrightarrow\\
&\qquad\qquad (f\ p\ s \Longleftrightarrow f\ (\lambda c_1\ \_ \to c_1 \in \gamma_{ST}\ (f^{\#}\ s^{\#}) \land p\ c_1\ ())\ s)
\end{aligned}
$$

More specifically, for any instruction `i`, the abstract interpretation $(\mathsf{wp}^{\mathsf{stmt}}_{\mathsf{w}}\ i)$ of `i` should be sound w.r.t. its regular weakest-precondition $\mathsf{wp}^{\mathsf{stmt}}_{\mathsf{w}}\ i$.

Theorem 1 *For any statement* `s:stmt`*,* $\mathsf{asem}_{\mathsf{stmt}}$ `s` *is sound with respect to* $\mathsf{wp}^{\mathsf{stmt}}_{\mathsf{w}}\ i$.

Proof 1 *By composition of the soundness of* $\mathsf{W}^{\#}$ *w.r.t. IMP*$^{\mathsf{x}}$*'s semantics and the fact the weakest-precondition calculus formed by the monad* `W` *implements IMP*$^{\mathsf{x}}$*'s semantics.*

4.5.3 Statement of Soundness

This section presents Theorem 2, our theorem of soundness. It states that one can confidently prefer to prove a specification on a program using its computed hybrid proof obligation, instead of proving the original, more complicated, proof obligation.

Theorem 2 (The hybridization W^h is sound) *For any program* `prg` *of type* `stmt`*, any pre-condition* `pre` *of type* $\mathsf{pre}_{\mathsf{w}}$*, any post-condition* `post` *of type* $\mathsf{post}_{\mathsf{w}}$ **`unit`***, and any* $\alpha{:}$ `approx`*, the lightened proof obligation* PO_h α `prg pre post` *implies the original proof obligation* PO_{W} `prg pre post`.

$$
\begin{aligned}
&\forall\ (\alpha{:}\ \mathsf{approx})\ (\mathsf{prg}{:}\ \mathsf{statement})\ (\mathsf{pre}{:}\ \mathsf{pre}_{\mathsf{w}})\ (\mathsf{post}{:}\ \mathsf{post}_{\mathsf{w}}\ \mathbf{unit}).\\
&\qquad \mathrm{PO}_h\ \alpha\ \mathsf{prg}\ \mathsf{pre}\ \mathsf{post} \Longrightarrow \mathrm{PO}_{\mathsf{W}}\ \mathsf{prg}\ \mathsf{pre}\ \mathsf{post}
\end{aligned}
$$

4.6 Proof Overview

This Section aims at giving some intuition about the way our proof is conducted. The full details are available as F* code. The statement of Theorem 2 holds for every program of type `statement`. The building blocks of our hybrid weakest precondition monad W^h are the different combinators of Section 4.4. Most of the proofs consist in proving that those combinators admit certain properties. Before looking at which property we are interested in, let us understand the meaning, definition and verification of a property about a weakest-precondition combinator.

At the end of Section 4.3.2, we presented how weakest-preconditions of type wp_{w} **`unit`** can be seen as maps from an initial memory to a set of admissible final memories. More generally, the function f_{wp} of type $\mathsf{wp}_{\mathsf{mon}}\,\mathsf{st}\,\mathsf{w}\,\tau$[5]

[5] With `st` a state type, `w` a type transformer for wrapping results and τ the type of the computation in stake

is a weakest-precondition but can also be seen as a map from states `st` to the Cartesian product of states `st` and outcome values of type τ. Indeed, the expression $\lambda s_0\ s \rightarrow f_{wp}$ (curry s) s_0[6] is of type `st`$\rightarrow$(`st`$\times\tau$)$\rightarrow$`prop`. In this section, we focus on the properties that the semantics reflected by a weakest-precondition yields. By abuse of language, we will refer to the initial and final states (or final outcome values) a weakest-precondition admits.

[6] With `curry` being the function defined below.

```
let curry (f:(τ×β)→γ): τ → β → γ
  = λx y → f (x,y)
```

4.6.1 Reasoning on Weakest-Precondition: "Meta" Hoare Triples

4.6.1.1 A First Example: purity

Our language IMP^x separates expressions and statements making sure expressions are pure, i.e. have no effect on the memory store. Thus, among all possible weakest-preconditions, it is interesting to select the ones that reflect a pure semantics. Consider f_{wp} : `wp st w` τ, with arbitrary `st`, `w` and τ. For every initial state s_0:`st`, and post-condition `p`:`post st w` τ, the post-condition `q` defined as $\lambda s_1\ \_ \rightarrow s_0 == s_1$ should be given for free by f_{wp}. That is, f_{wp} p s_0 should imply f_{wp} (p$\oplus$q) s_0. The expression p $\oplus$ q denotes the conjunction of p and q. The definition of operator $\oplus$ is given below.

```
let (⊕) #t #st #w (p q: post st w t)
  : (r: post st w t {order_post r p ∧ order_post r q})
  = λs r → p s r ∧ q s r
```

We generalize this kind of reasoning by introducing "meta" Hoare triples, i.e. a pre- and a (meta) post-condition about a weakest-precondition. A meta post-condition is a post-condition indexed by an initial state. The predicate `respects p` f_{wp} `mq` states that f_{wp} always yield post-condition mq s_0 for free given when p s_0 holds. It is then easy to write down the refined type $\texttt{constant}_{wp}$, that is inhabited by the weakest-preconditions of pure computations.

```
type post_meta st w (t: Type) = st → post st w t
let respects #st #w
  (p:pre st) (f: wp' st w τ) (q:post_meta st w τ): prop
  = ∀ (r: post st w τ) s.
          p s ⟹ f r s ⟹ f (q s ⊕ r) s
let constant_mp #st #w #t: post_meta st w t = λs0 s1 _ → s0 == s1
type constant_wp #st #w #t
  = f: wp' st w t {respects pre_top f constant_mp }
```

Let us now take a tour of some of the meta properties we are interested in for our proof of soundness.

4.6.1.2 State Consistence

We call a hybrid state ($s^\#$, s) consistent whenever s is approximated by $s^\#$, that is, s $\in$ $W^\#.\gamma$ $s^\#$. This is illustrated by Figure 4.8. This notion is implemented as a meta pre-condition and post-condition below.

```
let consistentSt_pre: pre st
  = λ(s#_0, s_0) → s_0 ∈ s#_0
let consistentSt_mp : post_meta st hVal τ
  = λ_ (s#_1, s_1) _ → s_1 ∈ s#_1
```

It is easy to state that f_{wp} (from Figure 4.8) is a weakest-precondition that preserves consistency (i.e. starting with an initial consistent state leads to final states that are consistent as well), by using the predicate `respects`. It amounts to using the pre- and post-conditions we just defined, as following: `respects consistentSt`$_{pre}$ f_{wp} `consistentSt`$_{mp}$.

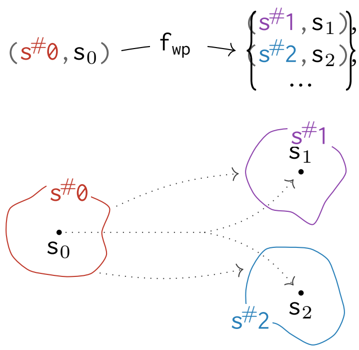

Fig. 4.8: Illustration of state preservation of *consistence*. Here, the weakest-precondition f_{wp} , given a *consistent* initial hybrid state (i.e. $(s^{\#}_0, s_0)$ with $s_0 \in s^{\#}_0$), yields hybrid state *consistence* as a free post-condition.

4.6.1.3 Weakest-Preconditions Respecting Given Abstract Interpretations

Below we define the type `wp_prop_h`, that holds the hybrid weakest-preconditions whose abstract semantics coincides with some abstract interpretation. The hybrid meta post-condition `eqAbSt`$_{mp}$ $f^{\#}$ states that abstract final states s_1 should be exactly $f^{\#}$ s_0. This post-condition is illustrated by Figure 4.9: the initial and final abstract state admitted by g_{wp} exactly corresponds to an abstract interpretation $g^{\#}$. Note that the weakest-precondition f_{wp} of Figure 4.8 does not respect one given abstract interpretation since an initial abstract state $s^{\#}_0$ admits two distinct abstract states $s^{\#}_1$ and $s^{\#}_2$.

Similarly to `eqAbSt`$_{mp}$, hybrid meta-post condition `eqAbExp`$_{mp}$ $r^{\#}$ states that the abstract component of outcome values should be exactly $r^{\#}$.

```
let eqAbSt_mp #t (f#: A.t): post_meta st hVal t
  = λ(s#_0,_) (s#_1,_) _ → s#_1 == f# s#_0
let eqAbExp_mp (r#: exp τ): post_meta st hVal τ
  = λ_ _ (r#',_) → r# == r#'

let wp_prop_h τ
  = f: wp_mon st hVal τ {
        (∃ f#. respects pre_top f (eqAbSt_mp f#))
      ∧ (∃ r#. respects pre_top f (eqAbExp_mp r#))
      }
```

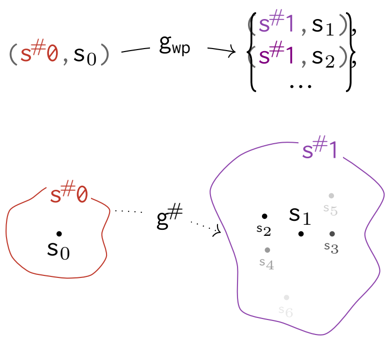

Fig. 4.9: Illustration of a hybrid weakest-precondition f_{wp} respecting an abstract interpretation $f^{\#}$.

Bind Below we define the lemma `bind_respects`. It states that the hybrid bind operation transports meta Hoare triples.

```
let bind_respects #u #v
  (f_pre: pre st) (f: wp_h^mon u) (f_post : post_meta st hVal u)
  (g: hVal u → wp_h^mon v) (g_post : st → hVal u → post_meta st hVal v)
  : Lemma (requires respects f_pre f f_post
                  ∧ (∀ s_0 r_1. respects (λs_1 → f_post s_0 s_1 r_1)
                                         (g r_1)
                                         (g_post s_0 r_1)
                    )
          )
```

```
          (ensures respects f_pre
                          (bind_wp_mon_h f g)
                          (λs₀ s₂ r₂ → ∃ s₁ r₁.
                                 f_post s₀ s₁ r₁
                               ∧ g_post s₀ r₁ s₁ s₂ r₂
                          )
          )
  = ⋯
```

This lemma can be used to propagate properties in a bind operation. Reconsidering our example of pure computation, the lemma makes it easy to prove if x_{wp} is pure, then $plus_5\ x_{wp}$ below is pure as well. Below, lemma xy_lemma makes a trivial use of bind_respects to encode the preservation of purity we discussed.

```
let plus₅ (x_wp : wp_h_mon ℤ): wp_h_mon ℤ
  = bind_h x_wp (λx => 5 + x)
let xy_lemma (x_wp : wp_h_mon ℤ)
  : Lemma (requires respects pre_top x_wp constant_mp )
          (ensures  respects pre_top (plus₅ x_wp ) constant_mp )
  = bind_respects pre_top x_wp constant_mp plus₅ constant_mp
```

4.6.2 Per Weakest-Precondition Soundness

While the statement of Theorem 2 operates on statements, below we define a statement indexed with both a regular and a hybrid weakest-precondition, soundE. Given a hybrid precondition pre, a hybrid weakest-precondition f_h and a regular weakest-precondition f_w, soundE pre f_h f_w states that for any regular post-condition p and state s_0 approximated by abstract state $s^{\#}_{\emptyset}$, if the precondition holds, then the hybrid proof obligation with the post-condition p via should be at least as strong as the regular one.

```
let soundE (pre: pre_h) (f_h: wp_h^mon τ) (f_w: wp_w^mon τ): prop
  = ∀ (p: post_w τ) s#_∅ s₀.
       pre (s#_∅, s₀)
    ⟹ s₀ ∈ γ s#_∅
    ⟹ f_h (λ(_,s₁) (_,r) → p s₁ r) (s#_∅, s₀)
    ⟹ f_w p s₀
```

Bind. For example, below bind_soundE carries soundness in the bind operation. Given a hybrid f_h that yields the post-condition f_{post} given the pre-condition f_{pre} holds, below ❶ states that f_{post} should entail $consistentSt_{mp}$. The proof that f_h is sound w.r.t. f_w (and similarly for g_h and g_w) is transformed by bind_soundE into a proof that the hybrid bind of f_h with g_h is sound w.r.t. $bind_w\ f_w\ g_w$.

```
let bind_soundE
  (f_h: wp_h^mon τ) (g_h: hVal τ → wp_h^mon β)
  (f_w: wp_w^mon τ) (g_w: id τ → wp_w^mon β)
  (f_pre: pre st) (f_post: post_meta st hVal τ)
```

```
  : Lemma (requires soundE f_pre f_h f_w
                   ∧ respects f_pre f_h f_post
                   ∧ ❶ order^META_post f_pre f_post consistentSt_mp
                   ∧ (∀ s0 r1. soundE (λs1 → f_post s0 s1 r1)
                                      (g_h r1)
                                      (g_w (snd r1))
                     )
          )
          (ensures  soundE f_pre (bind_wp_mon_h f_h g_h) (bind_w f_w g_w))
```

4.6.3 Proving Soundness

The proof of Theorem 2 is conducted by induction on statements. In Section 4.5.1.2, the function $\mathrm{wp}_h^{\mathrm{stmt}}$ constructs a hybrid weakest-precondition of type $\mathrm{wp}_h^{\mathrm{mon}}$ **unit** given a statement. Below, we provide the type signature of $\mathrm{swp}_h^{\mathrm{stmt}}$, that produces state consistency preserving weakest-preconditions, of type wp_prop_h. This type ensures there exists an abstract interpretation encoding the very initial to final abstract memory mapping for each inhabited weakest-precondition. The refinement also mentions that the produced weakest-preconditions are sound w.r.t. their regular counter-part.

The implementation of $\mathrm{swp}_h^{\mathrm{stmt}}$ consists in applying the different properties we discussed in this section on the various combinators involved.

```
let rec swp_h^stmt (i: stmt)
  : r: wp_prop_h unit
      { soundE pre_top r (wp_w^stmt i)
      ∧ respects consistentSt_pre r consistentSt_mp  }
  = ···
```

Function $\mathrm{swp}_h^{\mathrm{stmt}}$ is actually a slight reformulation of Theorem 2. Statement-wise $\mathrm{swp}_h^{\mathrm{stmt}}$ produces the same weakest-preconditions as $\mathrm{wp}_h^{\mathrm{stmt}}$.

4.7 Generalization: a Dijkstra Monad Transformer

We demonstrated the realization of our hybridization (Section 4.4) on a given input (Section 4.3). This section aims at showing how this instance of hybridization can be generalized so that it can be applied to other weakest-precondition monads and abstract interpreters.

First, we show how to fill out the blanks left in Sections 4.2.1 and 4.2.2: more precisely, below, Section 4.7.1 gives a proper type for the fields e_actions and ab_action from the record type $\mathrm{monad}_{\mathrm{wp}}$ and abint. Second, Section 4.7.2 shows how these action-related fields can be used to generically produce hybrid actions.

In this section, we consider a couple of weakest-precondition monads and abstract interpreters (e, ai), of type ($\mathrm{monad}_{\mathrm{wp}}$ × abint).

4.7.1 Actions

In this generalization, we only consider actions that have no impact on control flow; we disregard conditionals or loops. This can indeed be observed in the definition of record types $\texttt{monad}_{\texttt{wp}}$ and `abint` given in Sections 4.2.1 and 4.2.2: they have fields for certain fixed operations (i.e. `bind`, **`if`**, `while`).

Dereference operator and assignment are actions that cannot alter the control flow of the program by themselves. For simplicity, we only consider either pure actions whose outcome might be informative (e.g. addition), or impure actions whose outcome is non-informative (e.g. assignment). This distinction is encoded by the type $\texttt{action}_{\texttt{kind}}$. The type $\texttt{action}_{\texttt{type}}$ is inhabited by descriptions of the arrow type of an action.

```
type action_kind =
  | Exp: output:Type → action_kind
  | Stmt: action_kind
type action_type = list Type × action_kind
```

It is then possible to write a type-level function that transforms such an action type description into an actual arrow type: that is what `action_wp_type` does below. It is indexed by a weakest-precondition monad transformer and an action type description, and its outcome is a weakest-precondition respecting the monad in stake. For instance, the description $([\mathbb{Z};\mathbb{Z}],$ **`Exp`** $\mathbb{Z})$ is transformed into $\texttt{e.w}\ \mathbb{Z} \to \texttt{e.w}\ \mathbb{Z} \to \texttt{wp}_{\texttt{mon}}\ \texttt{e.st}\ \texttt{e.w}\ \mathbb{Z}$ by `action_wp_type e` considering `e` a monad.

```
let action_wp_type: monad_wp → action_type → Type
let action_abint_type: abint → action_type → Type
```

In a very similar way, `action_abint_type` transforms a type description into an abstract interpretation type.

The type of the field `e_actions` of a monad `e` consists in a list of descriptions and implementations of actions. A weakest-precondition action $\texttt{action}_{\texttt{wp}}$ `e` gathers a type description with a corresponding implementation. The exact same process goes for abstract interpretation as well, with the type $\texttt{action}_{\texttt{ai}}$.

```
type action_wp (e: monad_wp) =
  | Action_wp : type_desc: action_type
           → implem: action_wp_type e type_desc
type action_ai (ai: abint) =
  | Action_ai : type_desc: action_type
           → implem: action_ai_type ai type_desc
```

4.7.2 Generic Hybridization

Consider the type description $([\texttt{t}_0;\cdots;\texttt{t}_n],$ **`Stmt`**$)$, a weakest-precondition monad `e` equipped with an action $\texttt{action}_{\texttt{e}}$ of type $\texttt{e.w}\ \texttt{t}_0 \to \cdots \to \texttt{e.w}\ \texttt{t}_n \to \texttt{wp}\ \texttt{e.st}\ \texttt{e.w}$ **`unit`**, and an abstract interpreter `ai` equipped with an action $\texttt{action}_{\texttt{ai}}$ of type $\texttt{ai.exp}\ \texttt{t}_0 \to \cdots \to \texttt{ai.exp}\ \texttt{t}_n \to \texttt{ai}.\mathbb{M}^\# \to$

ai.$\mathbb{M}^{\#}$. Then, the hybridization of $\texttt{action}_{\texttt{e}}$ and $\texttt{action}_{\texttt{ai}}$ is defined as below. Note it is very similar to $\texttt{alloc}_{\texttt{h}}$ defined in Section 4.4.2.

```
let actionₕ (x₀: e.w t₀) ⋯ (xₙ: e.w tₙ): wp_mon unit
= λ(x₀, x#₀) ⋯ (xₙ, x#ₙ) →
    λp (s₀, s#₀) →
      let s#₁ = action_w x₀ ⋯ xₙ s#₀ in
      action_w x₀ ⋯ xₙ (λs₁ r →
        s₁ ∈ ai.γ s#₁ ⟹ p (s₁, s#₁) ((), ()#)
      ) s₀
```

The process of generalizing pure actions is very similar to the one above, following the definition of $\texttt{deref}_{\texttt{h}}$ given in Section 4.4.2.

4.8 Related Work

The burden of annotating F⋆ programs has been addressed in numerous ways. Low* [Pro+17] model memory as hyper-stacks, enabling modular region-specific and hence lighter invariants to be specified. Monotonic states [Swa+13; Ahm+18] facilitate the expression of invariants that are preserved over time, reducing the need for explicit invariants. Steel-Core [Swa+20] is a concurrent separation logic in F⋆ that makes, among others, concurrency-related invariants easier to express. These approaches ease the formulation of invariants in a specific use case; instead, our hybridization *infers* invariants directly.

K. Maillard et al [Mai+19] develop a powerful framework for manipulating Dijkstra monads, and focus on monad morphisms from computational to specificational monads. This allows to extend the scope of language features in F⋆ (non-determinism, IO) in a unified correctness framework. In our work, extending computational monads is irrelevant because our only aim is to lighten proof obligations by transforming specificational monads. Consequently, we did not leverage K. Maillard's framework in our soundness proof. Our statement of soundness amounts to a simple implication between proof obligations.

R. Jhala et al [JMR] introduce a verification procedure for higher-order functional programs using static analysis designed for imperative languages and leveraging refinement types. They translate refinement constraints about high-order programs as first-order programs: these can then be analyzed by a regular abstract interpreter, thus reducing the need for annotations.

A. Ivašković et al [IMO20] embed a control-flow analysis into a type system, using graded monads, in a non-dependent type setting. Instead, we directly embed an abstract interpreter in the Dijkstra monads of a dependent type system.

Liquid Types (Logically Quantified Data Types [MKJ08]) enable a restricted but decidable form of dependent type checking. Liquid Types leverage abstract interpretation to seek the strongest refinement satisfying a set of constraints. Liquid Haskell [VSJ14] is a static type checker that brings refinement types to Haskell, using Liquid Types independently of Haskell type checking. This approach has the benefit of automation, requires few (function type signature) annotations, at the cost of a relatively weak specification logic (QF-EUFLIA [VSJ14]). Dijkstra monads allow for much higher expressiveness, but yield a requirement of heavy annotations that our work strives to reduce using abstract interpretation.

4.9 Conclusion and Future Work

We introduce a method to embed abstract interpretation in a weakest-precondition calculus by transforming Dijkstra monads. This hybridization lightens the amount of both required annotations and generated proof obligations of the calculus. It is supported by the implementation of a working prototype in the dependently-typed language F* and a proof of soundness. Our implementation is purposefully a proof of concept: it models a simple Dijkstra monad, implements a simple abstract interpreter to run a simple IMP$^{\times}$ language, for the purpose of demonstrating the key concepts of our method. One current important limitation to the concrete use of our approach is our conditional combinator, that forks abstract memories in an exponential manner; consequently our approach currently disappoints our expectations in terms of applicability.

Currently, our method relies on a basic abstract interpreter that infers numerical intervals for program variables. To handle, e.g., C constructs, we would need to incorporate abstraction techniques used in Verasco or Astrée to track properties such as liveness of memory frames, alignments, and aliases. Similarly, our prototype transforms Dijkstra monads, not actual F* definitions in its effect system. Layered effects [Ras+21] make effects more flexible and expressive, and would make our hybridization easier to implement on actual F* effects. Having actual F* effect transformations and full C abstract interpreters is an attractive direction for future works which, we believe, would be a valuable contribution to F*/Low* community.

CHAPTER 5

LIO*: Dynamic and Static IFC policies

Contents

Chapter 3 showcased a straightforward and accessible implementation of a verified sound abstract analyser. Such a verified analyser enables trustworthy automatic formal analysis of programs. Then, Chapter 4 proposed a monad transformer leveraging such sound abstract analysers in order to ease verified programming. The scope of properties that these two chapters aim at verifying is relatively general. In this chapter, we investigate the opposite approach by picking one specific kind of properties to analyze.

This chapter is interested in *Information Flow Control* (IFC) policies. It implements an F⋆ variant of Labeled Input Output (LIO) [Ste+11] (Sections 5.4 and 5.5), a monadic IFC Haskell library. Our library intends to ease specification and verification of IFC policies for F⋆ and Low⋆ programs. We investigate the full spectrum of such policies, from fully static (Section 5.4) to fully dynamic (Section 5.5) verification. The clients of our library enjoy the compatibility of our library with Low⋆, enabling their extraction to efficient C programs[1] (See Sections 5.4.6 and 5.6). Leveraging Low⋆ and KreMLin (a tool for extracting F⋆ to C code, see Section 2.3.3), our library is well-suited for software aimed at low-level, embedded and/or resources-constrained devices. We also propose a method to formulate and prove noninterference theorems using meta-programming (Section 5.7).

[1] As explained in Chapter 2, in order to be extracted to C, the clients of our library should however be written in the Low⋆ subset of F⋆.

5.1 Introduction

The software systems that surround us are very often composed of multiple different components. For instance, consider a car. As illustrated by Figure 5.1, the on-board computer of a car acts as an orchestrator; it receives and sends information from and to various components of very different sensitivity levels. In this context, an example of an information flow property is that no brake-related decision should be taken by the on-board computer based on data emanating from the car radio component.

An IFC system tracks the various bit of data fed into a software along its lifespan. Each piece of information being tracked, it is possible to verify whether its flow respects a given policy. An example of IFC policy is isolation, i.e. a secret should never interact with a given portion of code. Such isolation policies meet security concerns: hence information flow policy is a broad and well studied topic. IFC systems are either dynamic or static: either the *control* of the flow happens at runtime, or a type system ensures it. In the former case, the runtime representation of data is enriched with a label, tracking, e.g., whether the data is secret or public. In the latter case, it is the type system itself that tracks such meta-information, and leaves the runtime representation of data untouched. One downside of a system that ensures an information flow policy at runtime is its overhead in terms of memory and computations. Moreover, it leaves room for policy-related failures: a program that violates a policy will yield an exception or terminate. Those two issues are particularly bothering in the case of critical embedded devices.

On the other side, consider a program that manipulates data of arbitrary sensitivity levels. To get some compile-time knowledge of the data at stake, the developer has to write tests to discriminate its sensitivity. Certain use-cases simply yield so many such tests that we end up with a runtime overhead comparable to the one implemented by a dynamic IFC library, without the practicality of such a library. In such cases, a dynamic library is better suited.

So, what is the better: dynamic or static IFC systems? It largely depends on the needs. The different tasks performed in a same program might meet very different needs. Whence our library, that lets the programmer choose at any point the nature of the IFC policy enforcement, from static to dynamic.

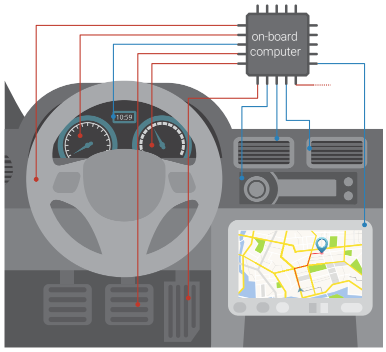

Fig. 5.1: A car on-board computer deals with different components that should not interact with each other in an arbitrary way. For instance, the blue-linked components shall not interact with red ones.

5.2 Labeling Information

In order to verify information flow policies, we have to keep track of the roles played by the various pieces of data at stake in a program. To do so, we label the various values we deal with, by wrapping them into *labeled values*. The type of labeled values `lv` is indexed by a label type and a value type. Consider the enumeration type `type label = | Secret | Public` which allows us to differentiate values that are public from the ones that shall stay private. An integer labeled with a tag `Secret` or `Public` has the type `lv label` $\mathbb{Z}$. Importantly, the type `lv` has no public constructor and cannot be destructed: one shall not unlabel a secret protected by a "private" label (for instance `Secret`) and, e.g. shall send it over the internet. The unwrapping of such labeled values must be *controlled*.

5.2.1 Hiding type constructors

By default, in F*, a module *exports* all its definitions: a client to a module can see all its implementation details. The mechanism of *module interfaces* however allows a module to be split into an implementation and an interface. An interface can contain signature declarations without implementation, as well as implemented definitions. A client to a module equipped with an interface is blind to the declaration from the implementation, and only sees the interface.

In order to control construction and unwrapping of labeled values, in the interface of the module providing labeled values, we only declare the signature of lv without any constructor. Figure 5.2 presents this interface. Given a label type lt and a value type a, lv lt a is the type of labeled a values, with labels of type lt. Notice that the type signature of lt is a type transformer with explicit type-universes (See Section 2.1.3). Given a label type of universe u#lt and a value type of universe u#a, labeled values are of type whose type universe is u#(max lt a).

```
val lv (lt: Type u#lt) (a: Type u#a): Type u#(max lt a)
val labelOfₑ (v: lv τ β): Ghost.erased τ
val valueOfₑ (v: lv τ β): Ghost.erased β
let labelOf (v: lv τ β): GTot τ = Ghost.reveal (labelOfₑ v)
let valueOf (v: lv τ β): GTot β = Ghost.reveal (valueOfₑ v)
val ghost_make (l:τ) (v:β)
  : GTot (r:lv τ β {labelOf r == l ∧ valueOf r == v})
```

Fig. 5.2: Interface for the module providing labeled values.

5.2.2 Computational Relevance

The observation of the label or content of a labeled value is possible via the functions $\texttt{labelOf}_e$ and $\texttt{valueOf}_e$. Such observations are computationally-irrelevant: functions[2] $\texttt{labelOf}_e$ and $\texttt{valueOf}_e$ both produce values of type of the shape **Ghost**.erased ϵ, with some ϵ. Such values of type **Ghost**.erased ϵ are isolated from the world of informative values. The type **Ghost**.erased ϵ wraps values of type ϵ in a box that, virtually, has () (the inhabitant of **unit**) as runtime representation: no decision can be derived from ().

The only interface provided for unwrapping erased value is the function **Ghost**.reveal. For any type τ, it has the arrow type erased $\tau \to$ **GTot** τ: it transforms an erased value into a non-informative computation. As explained in Section 2.3.1, the effect **GTot** acts as a sink: information emanating from a **GTot** computation is marked non-informative and cannot be used in a computationally-relevant context. Our goal being to control how information flows to avoid leakage at runtime, it is fine to represent labels and values at type-level.

Functions $\texttt{labelOf}_e$ and $\texttt{valueOf}_e$ live in the **Tot** effect, but they produce erased values. By contrast, functions labelOf and valueOf live in the **GTot** effect of non-informative computations, but produce plain values. It is often more convenient to work directly with unwrapped values, whence these ghost computations.

[2] Recall that an arrow type $\tau \to \beta$ is a shortcut for $\tau \to$ **Tot** β. The effect-explicit type of $\texttt{labelOf}_e$ is thus v:lv τ $\beta \to$ **Tot** (**Ghost**.erased τ).

5.2.3 A Zero-Cost Abstraction

Now, let us take a look at Figure 5.4, presenting the –hidden by means of a module interface– implementation of our labeled value module. Note that the runtime representation of a labeled value of type lv τ β is isomorphic to β, since **Ghost**.erased τ is computationally irrelevant. Via a few F* attributes and qualifiers[3], it is easy to instruct KreMLin to completely eliminate the record type lv τ β: every labeled value of type lv τ

[3] The module abstraction allows to e.g. hide type constructors for verification purposes. However, it is possible to teach F* to drop such modular abstractions while extracting F* code to C or OCaml. Also, the **noextract** attribute teaches KreMLin it should not extract a specific definition. Other methods can be used to eliminate useless abstractions, i.e. normalizing certain definitions when extracting.

β is simply regarded as a plain value of type β. Figure 5.3 illustrates this elimination. Similarly, KreMLin eliminates every call to `labelOf`$_e$, `valueOf`$_e$, `labelOf`... The entire module related to labeled values is actually eliminated during KreMLin's extraction to C. In consequence, we cannot illustrate how this module is translated to C, as it is outright dropped.

```
type lv lt a = { lbl: Ghost.erased lt; v: a }
let labelOf_e v = v.lbl
let valueOf_e v = v.v
let ghost_make l v = { lbl = l; v = v }
let trusted_make (l: Ghost.erased τ) (v: τ) = {lbl = l; v = v}
```

Fig. 5.4: Hidden implementation for the module providing labeled values.

Also note the definition `trusted`$_{make}$: the reader might wonder what its purpose is since it is not exported in the interface of the module (Figure 5.4), and hence remains invisible. F* modules can have *friend modules*, sharing their hidden definitions. Section 5.4.4 will present how our main IFC effect makes use of this feature.

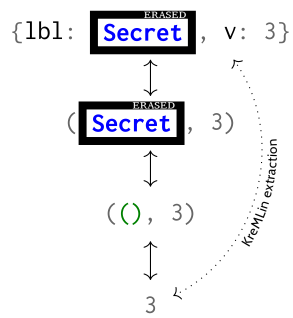

Fig. 5.3: The computable representation of a labeled value is isomorphic to its wrapped value. Two values connected by an arrow are isomorphic in terms of runtime representation.

5.2.4 Values With a Runtime Label

In various scenarios, the label protecting a value is part of the runtime data. Consider for example a private note web application: its database would contain a table of notes, where each row stores a note of type **string** along with a user identifier of type e.g. `user` = | **Bob** | **Alice** | **Eve**. Let us consider the lattice presented in Figure 5.5, taking `set user` as label type. One can represent such rows by considering different type representations. A first approach would be to represent a row as a `lv user` **string**. This however amounts to discarding the column "user": the label of a labeled value `lv` disappears at runtime. A solution is then to encode a row as a tuple `user` × `lv user` **string**. On a type-level point of view, such a tuple type is disappointing: it allows for tuples whose runtime label is not equal to the type-level label held in the labeled value.

Our library provides the type `lv`$_{rt}$ to represent labeled values with a runtime representation, and to ensure the consistency between type-level and runtime labels. Its field `v`$_{rt}$ is refined so that `lbl`$_{rt}$ is always equal to `v`$_{rt}$'s type-level label.

```
type lv_rt τ β = {lbl_rt :τ; v_rt : x:lv τ β {labelOf x==lbl_rt }}
```

This section presents both statically and dynamically labeled values. Dynamically labeled values proxy the protection of statically labeled values. However this section does not present how one deals with such protected values: the protected values cannot (yet) be constructed or unwrapped. Before diving in the monadic behavior of LIO (Section 5.4), which enables labeled value manipulation, the next section looks at which kind of structure labels form.

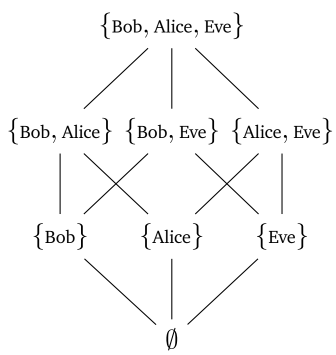

Fig. 5.5: The lattice (`set user`, ⊆) formed by sets of users and ordered by inclusion.

5.3 Labels as a Lattice

Labeled values allow us typically to track the security level of values dealt within a program. Let `l` the type of labels; i.e. the enumeration `type l = | Low | Medium | High` representing three security levels. As soon as we start mixing values together– i.e. concatenating two values– the security levels should be mixed as well. Some security labels are *higher* than others: reading a value labeled `Low` is fine in a context dealing with `High` values. The fact a value (protected by a certain label) *can flow* to a certain (labeled) context of security is decided by an order on labels, denoted $\sqsubseteq$. For instance `Low` $\sqsubseteq$ `Medium` means that `Low`-sensitive values can flow to a `Medium` context of security. We also consider $\sqcup$ the binary operation that mixes two labels together, that is $\sqcup$ is of type `l` $\rightarrow$ `l` $\rightarrow$ `l`. For consistency, the structure `(l,`$\sqsubseteq$`)` should form a join-semilattice, thus the join operator $\sqcup$ should compute least upper bounds [Den76]. Below, we define a new typeclass for join-semilattices, to which we refer simply as lattices in this chapter. Every inhabitant of `lattice` τ (for τ a type) shall provide proofs for $\sqsubseteq$'s reflexivity ($\texttt{refl}_{\texttt{ord}}$), transitivity ($\texttt{trans}_{\texttt{ord}}$) and anti-symmetry ($\texttt{antisym}_{\texttt{ord}}$). The field $\sqcup$ is a binary operator refined so that it is an upper bound $\sqsubseteq$-wise; moreover, $\texttt{join}_{\texttt{lub}}$ ensures $\sqcup$ computes least upper bounds.

```
class lattice a = {
  ⊑: a → a → bool;
  ⊔: x:a → y:a → r:a { x ⊑ r ∧ y ⊑ r };
  refl_ord : l:a → Lemma (l ⊑ l);
  trans_ord : x:a → y:a → z:a →
    Lemma (requires x ⊑ y ∧ y ⊑ z) (ensures x ⊑ z);
  antisym_ord: x:a → y:a →
    Lemma (requires x ⊑ y ∧ y ⊑ x) (ensures x == y);
  join_lub: x:a → y:a → l:a →
    Lemma (requires x ⊑ l ∧ y ⊑ l) (ensures (x ⊔ y) ⊑ l);
}
```

We define two auxiliary extrinsic lemmas $\texttt{refl}_{\texttt{smt}}$ and $\texttt{trans}_{\texttt{smt}}$, that introduce SMT patterns. Thanks to them, the SMT solver will automatically instantiate reflexivity and transitivity for $\sqsubseteq$ operators when used.

```
let refl_smt (l: lattice τ) (x: τ)
  : Lemma (x ⊑ x) [SMTPat (x ⊑ x)]
  = refl_ord x
let trans_smt (l: lattice τ) (x y z: τ)
  : Lemma (requires x ⊑ y ∧ y ⊑ z) (ensures x ⊑ z)
          [SMTPat (x ⊑ z); SMTPat (x ⊑ y)]
  = trans_ord x y z
```

We now enjoy a few tools to represent and work with labeled information. In the beginning of this section, we discussed security contexts. In the following section, we are going to see that such contexts can be defined in a monadic way, using F*'s layered effects.

5.4 GLIO*: A Static Monadic IFC System

We described how a security lattice of labels can be used to protect values by wrapping them as labeled values. In this section, we introduce the effect **GLIO**, that keeps a type-level track of the security context of computations. Such a fully static approach allows to verify flow information policies without any runtime cost and without any IFC-related failure at runtime. The downside of such an approach is the human time cost: one shall prove his program respects given policies.

Section 2.2.6 presented the concept of computational monads indexed by weakest-precondition monads. In F*, such indexed monads can be written as *layered effects*. This section continues the discussion of Section 2.2.6 by defining **GLIO**, a concrete layered effect implementing a static IFC system.

Similarly to LIO [Ste+11], the IFC context (of type `context`) of a **GLIO** computation consists in `cur`, a current label and `cle`, a *clearance*. The current label reflects the level of security of the computation going on: for instance, in a **Low**⊑**Medium**⊑**High** lattice, after reading a secret the current label of a computation would be **High**. The notion of clearance is useful to state that a component of a program should never reach a certain level of security. Consider a component that is expected to manipulate non-critical only pieces of information. A simple way to ensure the component never deals with values beyond **Medium** (for instance) is to set the clearance to λ`cur` → `cur` ⊑ **Medium**. This predicate holds on labels that are below **High** in the lattice. In other words, a clearance allows to forbid a computation to flow to certain labels in the lattice. A few clearance policies are illustrated by the red dots in Figure 5.6.

```
type context = { cur: Ghost.erased labelType
               ; cle: cle: (labelType → Type0) {cle cur} }
```

The current label `cur` is of type `Ghost.erased labelType`, thus current labels are computationally irrelevant, and erasable by KreMLin. The type `labelType` and its lattice are implemented by a per-client module[4] **Parameters**. The field `cle` is a map from `labelType` to $\mathtt{Type}_0$. $\mathtt{Type}_0$ is inhabited by computationally irrelevant values: non-decidable predicates only have constructors, one cannot discriminate or destruct them; in other terms, it is not possible to derive any sort of information from those values. A total map whose codomain is erasable is erasable as well; thus the values of type `context` can safely be erased by KreMLin. The extraction mechanism therefore eliminates any sort of IFC contexts. We do not want to consider contexts in which the current label is forbidden by the clearance: whence the refinement of the field `cle`.

[4] At the time of writing, a bug in the extraction of layered effects forbids us to have an effect indexed by a type used as a state. See issue 1879. Instead of having our effect parametrized by label type, we thus fix it in a F* module **Parameters**. A client defines its own **Parameters** module to be used with the library.

5.4.1 A Specification Monad for GLIO

Following Section 2.2.3, in order to define the effect GLIO, we first define a monad of specification whose representations are weakest-preconditions. A weakest-precondition for a computation whose outcome type is τ is a map from post-conditions $\texttt{post}_\texttt{t}\ \tau$ to pre-conditions $\texttt{pre}_\texttt{t}$. Just like in Chapter 4, weakest-preconditions should be monotonic, post-condition wise. Let $\texttt{wp}:\ \texttt{post}_\texttt{t}\ \tau \rightarrow \texttt{pre}_\texttt{t}$, $\texttt{p}:\texttt{post}_\texttt{t}\ \tau$ and $\texttt{q}:\texttt{post}_\texttt{t}\ \tau$. If p is stronger than q, then the pre-condition wp p should be stronger than wp q. This property is spelled out by $\texttt{wp}_\texttt{mon}$.

```
type pre_t = context → HST.st_pre
type post_t a = HST.st_post (a × context)
let wp_mon (wp: post_t τ → pre_t)
  = ∀ (p q: post_t τ).
        (∀ x m. p x m ⟹ q x m)
     ⟹ (∀ c m. wp p c m ⟹ wp q c m)
```

One last property we want for our weakest-preconditions is directly related to IFC. Atomically, a computation should never decrease its contextual current security label: a computation that has initially access to secret informations should not be considered as non-sensitive. Similarly, a computation should never make its own clearance more permissive. Thus, below, we define the order $\lll$ over contexts; $\subseteq$ being an order for clearances. Figure 5.6 illustrates how contexts are ordered on an example lattice. The predicate wp_ctx_increases wp states that for every post-condition p, wp p should be stronger (or actually equivalent when wp is monotonic) than wp q, with q being the same as p but ensuring the correct order between contexts. The type $\texttt{wp}_\texttt{t}$ refines maps from post-conditions to pre-conditions to form the type inhabited by well-formed weakest-preconditions to IFC computations.

The types $\texttt{HST.st}_\texttt{pre}$ and $\texttt{HST.st}_\texttt{post}$ are part of Low* library. As we will detail later, our library *lives* in a Low* effect. Consequently, a pre- or post-condition about a GLIO program is an extended pre- or post-condition about a computation living in a Low* effect. HST refers to F* standard library module FStar.HyperStack.ST.

```
let (⊆) (s0 s1: τ → Type0) = ∀ (x:τ). s0 x ⟹ s1 x
let (⋘) c0 c1 = c0.cur ⊑ c1.cur ∧ c1.cle ⊆ c0.cle
let add_ctx_increases (c0: context) (p: post_t τ): post_t τ
  = λ(x,c1) m1 → c0 ⋘ c1 ∧ p (x,c1) m1
let wp_ctx_increases (wp: post_t τ → pre_t)
  = ∀ (p: post_t τ) (c0: context) m0.
      wp p c0 m0 ⟹ wp (add_ctx_increases c0 p) c0 m0
type wp_t a = wp: (post_t a → pre_t) { wp_mon wp
                                      ∧ wp_ctx_increases wp }
```

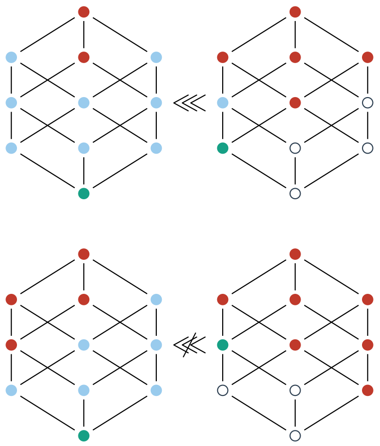

Fig. 5.6: Example of ordering between contexts. Each diagram represents the constraint that a context sets on the lattice at stake. The labels disallowed by the context clearance are red, the current label is green. Blue labels are the accessible labels.

The return operation $\texttt{return}_\texttt{wp}\ \tau$ x lifts $\texttt{x}:\tau$ in our weakest-precondition monad. $\texttt{return}_\texttt{wp}\ \tau$ x amounts to wrapping x in a continuation of type $\texttt{wp}_\texttt{t}\ \tau$. The bind operation $\texttt{bind}_\texttt{wp}\ \tau\ \beta$ f g is of type $\texttt{wp}_\texttt{t}\ \beta$: we bind the two weakest-preconditions and we inject invariants about increasing contexts.

```
let return_wp  (a:Type) (x:a): wp_t a = λp c0 m → p (x, c0) m
let bind_wp  (a:Type) (b:Type)
  (wp_f:wp_t a) (wp_g:a → wp_t b): wp_t b
  = λp c0 h0 → wp_f (λ(x, c1) h1 →
      c0 ⋘ c1
    ∧ wp_g x (λ(y,c2) h2 → c1 ⋘ c2 ∧ p (y,c2) h2) c1 h1
    ) c0 h0
```

The structure formed by $\texttt{return}_{\texttt{wp}}$, $\texttt{bind}_{\texttt{wp}}$ and $\texttt{wp}_\texttt{t}$ is our specification monad.

5.4.2 An Indexed Computation Monad for GLIO

We aim at defining an effect for fully static verification of information flow policies. Following [Ste+11], we use a monadic approach that consists in a state monad tracking the *current* security level thanks to an IFC context. Our effect **GLIO** consists in a layer above **STATE**, the principal effect of Low*. A **GLIO** computation `f` of type τ admitting `wp` as weakest-precondition is represented by an inhabitant of the type `repr` τ `wp`. More precisely, such a **GLIO** computation `f` is represented by a **STATE** computation that explicitly passes around a context state. Note that, from a computation point-of-view, `repr` τ `wp` is just a **STATE** computation without any context: `context` inhabitants are computationally irrelevant. Below we define the combinators `return` and `bind`. Notice their definitions are very straightforward: contexts being computationally irrelevant, no **GLIO** combinator (or computation) can derive any decision from them, thus their definition is rather canonical. The magic happens at the type level, in the second index of the representation type `repr`, where weakest-preconditions are computed via the specification monad we defined in Section 5.4.1.

```
let return (a:Type) (x:a): repr a (return_wp a x) = λc0 → x,c0
let bind (a b:Type) (wp_f:wp_t a)     (wp_g:a → wp_t b)
                    (f:repr a wp_f) (g:(x:a → repr b (wp_g x)))
  : repr b (bind_wp a b wp_f wp_g)
= λc0 → let (x, c1) = f c0 in g x c1
```

5.4.3 Effect definition

An F* (layered) effect **E** consists in an indexed monad along with a few other definitions: an effect gives to F* the various rules required to compute weakest-preconditions of computations in **E**. The definition `if_then_else` acts as an effect-wise typing rule for **if** ... **then** ... **else** ... constructions. The function `subcomp` teaches F* how subtyping works in our effect. In our case, subtyping is simple: a computation `f: repr a` $\texttt{wp}_\texttt{f}$ can be subtyped as `repr a` $\texttt{wp}'_\texttt{f}$ for any $\texttt{wp}'_\texttt{f}$ weaker than $\texttt{wp}_\texttt{f}$.

```
let if_then_else_wp (a:Type) (wpf wpg:wpt a)
  (p:eqtype_as_type bool): wpt a = if p then wpf else wpg
let if_then_else (a:Type) (wpf wpg:wpt a)
  (f:repr a wpf) (g:repr a wpg) (p: bool): Type
= repr a (if_then_else_wp a wpf wpg p)
let subcomp (a:Type) (wpf:wpt a) (wp'f:wpt a) (f:repr a wpf)
  : Pure (repr a wp'f)
         (∀ p c h. wp'f p c h ⟹ wpf p c h)
         (λ_ → ⊤)
  = f
```

Our actual `GLIO` effect is defined below. It is indexed by a type and a weakest-precondition (whence the type `a:Type→wpt a→Effect`).

```
reifiable reflectable layered_effect {
  GLIO : a:Type → wpt a → Effect
     with repr=repr; return=return; bind=bind;
     subcomp=subcomp; if_then_else=if_then_else
}
```

Before the keyword `layered_effect`, note the qualifiers `reifiable` and `reflectable`. As explained in Section 2.3.1, reflection (in our setting) is the process of transforming a computation of type `repr τ wp` into `GLIO τ wp`. Reification is the reverse process: given a `GLIO τ wp` computation, reification exposes its underlying representation `repr τ wp`. Figure 5.7 summarizes these processes. Obviously, such features partially defeat the policy enforcement of our IFC system, and are not to be used but for trusted IFC actions.

Before defining such actions, recall that F⋆ organises effects onto a lattice. Our effect is not marked with the qualifier `total`, and thus allows for divergence. This choice follows the effect `STATE`, the effect of the underlying representation of `GLIO`, that also allows for divergent computations. Regardless whether it terminates or it diverges, a pure computation can be lifted as a computation in `GLIO`. Using the syntax `sub_effect DIV ⇝ GLIO`, we let F⋆ automatically lift `DIV` computations into `GLIO`.

```
let lift_div a (wp:pure_wp a) (f:unit → DIV a wp)
  : repr ⋯ = ⋯
sub_effect DIV ⇝ GLIO = lift_div
```

As shown in Figure 2.7, `Tot` computations can be lifted as `DIV` computations: by transitivity, a pure total computation can thus be lifted as `GLIO` as well. We do not detail $\texttt{lift}_{\texttt{div}}$, that encodes `DIV` computations under `repr`, the representation type of `GLIO`.

Writing a specification in the form of a weakest-precondition is not intuitive, and leads to hard-to-understand specifications. This is why the F⋆ library comes with Hoare-style variants for its various effects. For example, `ST` is an Hoare-style synonym for the effect `STATE`. The function `glio_hoare_to_wp` constructs a weakest-precondition given a precondition `pre` and a post-condition `post`.

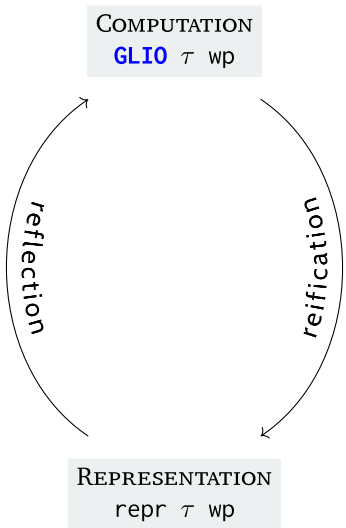

Fig. 5.7: Reification and reflection are two sides of a same coin, transforming a computation into its representation and vice-versa.

```
let glio_hoare_to_wp (#a: Type)
  (pre: pre_t)
  (post: ( c0:context          → m0:MHS.mem {pre c0 m0}
         → r:a
         → c1:context{c0⋘c1} → m1:MHS.mem
         → Type0
         ))
  : wp_t a
  = λ(p: post_t a) (c0:context) (m0: MHS.mem)
     → pre c0 m0
     ∧ (∀ (r: a) (c1: context) (m1: MHS.mem).
           (c0 ⋘ c1 ∧ post c0 m0 r c1 m1)
       ⟹ p (r, c1) m1
     )
```

The following declaration defines **GLio**, a synonym effect indexed by a type, a pre-condition and a post-condition.

```
effect GLio (a: Type)
  (pre: pre_t)
  (post: ( c0:context          → m0:MHS.mem {pre c0 m0}
         → r:a
         → c1:context{c0⋘c1} → m1:MHS.mem
         → Type0
         ))
  = GLIO a (glio_hoare_to_wp pre post)
```

5.4.4 IFC actions

So far, our effect **GLIO** is minimal: it only has a bind and return operation. Here, we define actions that manipulate IFC contexts and labeled values, following [Ste+11]. Remember **GLIO** is a layered effect: an effect whose underlying monad is an indexed monad. Each **GLIO** action thus comes in no less than three flavors: (i) an action in our specification monad of weakest-precondition (of type `wp_t ⋯`); (ii) an other one in our computation monad (of type `repr ⋯ ⋯`); (iii) and a last one, a **GLIO** computation (of type **GLIO** `⋯ ⋯`)..

GLIO computations are allowed to inspect the current IFC context, for specificational purpose only, via the action $\texttt{get}_{\texttt{ctx}}$. Similarly, $\texttt{get}_{\texttt{mem}}$ proxies **HST**`.get` from Low*, which allows Low* clients to inspect their memory model. Note the use of **GLIO**`?.reflect`: it enables us to re-interpret, i.e. $\texttt{get}_{\texttt{ctx}}$', as **GLIO** computations.

```
let get_ctx^wp : wp_t context = λp c h → p (c,c) h
let get_ctx' (): repr context get_ctx^wp  = λc → c, c
let get_ctx (): GLIO context get_ctx^wp
  = GLIO?.reflect (get_ctx' ())
let get_mem^wp : wp_t HS.mem = λp c h → p (h,c) h
let get_mem' (): repr HS.mem get_mem^wp  = λc → HST.get (), c
let get_mem (): GLIO _ get_mem^wp
  = GLIO?.reflect (get_mem' ())
```

It is always safe to *raise* the current label to the extent that the current clearance allows it. In a similar manner, one can always make the current clearance more restrictive. In other terms, one can always replace the current IFC context with a new one if it respects the order ⋘.

```
let set_ctx^wp (c1: context): wp_t unit
  = λp c0 h → c0 ⋘ c1 ∧ p ((),c1) h
let set_ctx' (c1: context): repr unit (set_ctx^wp c1)
  = λc → ((), c1)
let set_ctx (c1: context): GLIO unit (set_ctx^wp c1)
  = GLIO?.reflect (set_ctx' c1)
```

Labeled values The most interesting actions are related to labeled value manipulation. The `label` action lets a **GLIO** client label a value `v` at any label `l` as long as `l` is above the current label. Whence `c.cur`⊑`l` in the specification `label_wp` of `label`.

```
let label_wp (#a:Type) (l: Ghost.erased labelType) (v: a)
  : wp_t (lv labelType a)
  = λp c m → c.cur ⊑ l ∧ p (ghost_make (Ghost.reveal l) v,c) m
val label' (#a:Type) (l: Ghost.erased labelType) (v: a)
  : repr (lv labelType a) (label_wp l v)
let label (#a:Type) (l: Ghost.erased labelType) (v: a)
  : GLIO (lv labelType a) (label_wp l v)
  = GLIO?.reflect (label' l v)
```

Notice that there is no implementation given for `label'`: indeed, the constructor and destructor of the type `lv` are hidden (Section 5.2.1). To be able to construct labeled values, the module implementing **GLIO** actions is declared as a friend (See Section 5.2.3) of the one implementing labeled values. This friendship mechanism enforces isolation. From the definition of the type `context` (that is, begining of Section 5.4) to here, all the definitions we gave were exposed in the interface of the module **GLIO**. The public code present in the interface, even if friend with a labeled value module, cannot construct labeled values. `label` is defined in the *implementation* side of the module; we give it below. It simply leverages the `trusted_make` unsafe construct presented by Section 5.2.3.

```
let label' l v = λc → trusted_make l v, c
```

The `unlabel` primitive allows to unwrap labeled values, and follows the same rules as `label`: its actual definition is hidden. Unwrapping a labeled value causes the current label to be raised: this is what the specification action `unlabel_wp` encodes.

```
let unlabel_wp (#a:Type) (v: lv labelType a): wp_t a
  = λp c m → c.cle (c.cur ⊔ labelOf v)
    ∧ p (valueOf v, {cur = c.cur ⊔ labelOf v; cle = c.cle}) m
val unlabel' (#a:Type) (v: lv labelType a): repr a (unlabel_wp v)
let unlabel (#a:Type) (v: lv labelType a): GLIO a (unlabel_wp v)
  = GLIO?.reflect (unlabel' v)
```

The hidden implementation of `unlabel'` destructs the labeled value and performs a ghost join of the current label with the one specified by the labeled value at stake.

```
let unlabel' v
  = λc → v.v, { cur = c.cur ⊔ labelOf v
              ; cle = c.cle }
```

The last IFC core action is `toLabeled`. It is illustrated by Figure 5.8. It allows to run a computation without raising the current label or lowering the clearance. Instead, the context is preserved, and the eventual raise of the current label is captured by wrapping its outcome in a labeled value. Note that `toLabeled'` works with representations `repr ⋯` and not with `GLIO` computations. Consequently `toLabeled`, which takes a `GLIO` computation as input, reifies its input into a representation, so that `toLabeled'` can be called and its resulting computation be reflected as a `GLIO` computation.

```
let toLabeled_wp #a (wp_f: wp_t a): wp_t (lv labelType a)
  = λp c0 m0 →
    wp_f (λ(r,c1) m1 →
           p (ghost_make (Ghost.reveal c1.cur) r,c0) m1
      ) c0 m0
val toLabeled' (#a:Type) (#wp_f:wp_t a) ($f: repr a wp_f)
  : repr (lv labelType a) (toLabeled_wp wp_f)
let toLabeled #a #wp_f ($f: unit → GLIO a wp_f)
  : GLIO (lv labelType a) (toLabeled_wp wp_f)
  = let f: repr a wp_f = reify (f ()) in
    GLIO?.reflect (toLabeled' f)
```

The hidden implementation of `toLabeled'` runs the computation `f:repr` τ `⋯` it receives as input. The outcome of `f` is a tuple `(r,c1)`, with `r` of type τ and `c1` a context. The context current label `c1.cur` is used to wrap `r` as a labeled value. The context returned is not `c1`; this context `c1` is discarded, and the initial context `c0` is restored.

```
let toLabeled' f
  = λc0 → let r,c1 = f c0 in trusted_make c1.cur r, c0
```

5.4.5 A Basic Interface to Memory

Since our effect `GLIO` is represented as a Low* computation, we can easily proxy a subset of the memory model and API of Low*. Our library provides

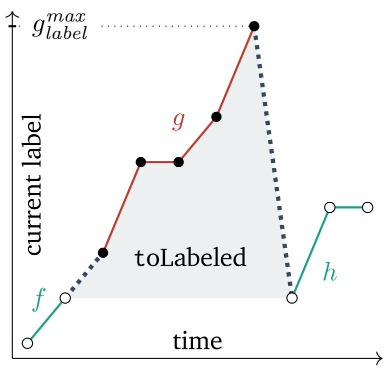

Fig. 5.8: Evolution of the current label for the following computation p, with some computations `f`, `g` and `h`.

```
let p (): GLIO ⋯
  f (); ❶
  let v:lv ⋯ = toLabeled g
  in ❷ h ()
```

The action `toLabeled` captures the label of `g` label and leaves the IFC context unmodified: `v` is labeled with g_{label}^{max}, and the current labels at ❶ and ❷ are the same.

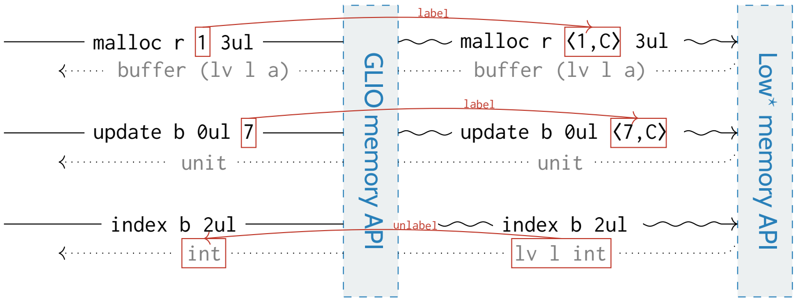

Fig. 5.9: **GLIO** bridges a small portion of the memory API of Low*. In this figure, **C** denotes the current label, `r` a region, and `b` a buffer. Allocating a buffer with **GLIO**'s `malloc` triggers an automatic labeling of the initial value the buffer is to be filled with. Updating a buffer forces a labelization as well, and dereferencing a label triggers an `unlabel` operation.

an interface for allocating, dereferencing and updating buffers in memory. Below we provide the simplified type signature for these primitives. The types **B**.`buffer` and **HS**.`rid` are provided by Low*; the former denotes pointers to buffers (that is, region of arbitrary size) while the latter denotes memory regions.

```
unfold let malloc (r:HS.rid) (v: τ) (len:U32.t)
  : GLio (B.buffer (lv labelType τ)) ⋯ = ⋯
let index (b:B.buffer (lv labelType τ)) (i: U32.t)
  : GLio τ ⋯ = ⋯
let upd (b:B.buffer (lv labelType τ)) (i:U32.t) (v:τ)
  : GLio unit ⋯ = ⋯
```

Our interface is designed to maintain IFC invariants: the memory-related actions make sure the data being stored in memory is always labeled. The memory API of **GLIO** forbids any access to non-labeled values stored in memory. Figure 5.9 illustrates how the API bridges memory operations to Low*.

5.4.6 An Example of GLIO computation

The program `ex` below has two arguments: `x` a pointer to a labeled integer and `y` a labeled integer. It dereferences the pointer `x` with `index`; `v` is thus an integer of type **U32**.`t`, that is a 32-bits unsigned integer. Then, it returns the addition of `v` with the value held in `y`, unwrapped with a call to `unlabel`. The pre-condition to `ex` ensures `x` points to a live region in memory, and requires (for the sake of simplicity) the clearance to authorize every single label. As a post-condition, we require that the label of the labeled value `y` is below the final current label. This is trivially true because here, `labelType` is the chain lattice $\mathbf{L} \sqsubseteq \mathbf{M} \sqsubseteq \mathbf{H}$.

```
let ex (x: B.pointer (lv labelType U32.t))
       (y: lv labelType U32.t)
: GLio U32.t
    (λc₀ m₀ → B.live m₀ x ∧ (∀ x. c₀.cle x))
    (λc₀ m₀ x c₁ m₁ → labelOf y ⊑ c₁.cur)
= let v = index x 0ul in
  U32.add_underspec v (unlabel y)
```

```c
uint32_t ex(uint32_t *x, uint32_t y)
{
  uint32_t x1 = x[0U];
  uint32_t x10 = x1;
  uint32_t x2 = y;
  return x10 + x2;
}
```

The code on the left is written in F*; the code on the right is the corresponding C code generated by KreMLin. Notice that the whole **GLIO** library is entirely erased by KreMLin; there is not a single definition left. The three useless assignments put apart, the C code of `ex` is very small as expected. The superfluous assignments are not problematic, as any modern C compiler will get rid of them.

If this library is purely specificational, this is not always desirable: building on **GLIO**, the next section will focus on another kind of **IFC** system, namely **DLIO**, a runtime-oriented **IFC** system.

5.5 DLIO*: A Dynamic IFC System as a GLIO* Client

In some scenarios, it is desirable to rely on a concrete representation of the current label. As discussed in Section 5.2.4, the label protecting a value is sometimes part of the data available at runtime. Similarly, the contextual label of a computation can actually be a useful piece of (runtime) information.

As an example, let us consider the following scenario in a school context. After an assignment, each student is given a mark. Consider the use-case where one wants to compute the means of different subsets of marks, and then sends the result to the correct person in charge of that subset. A set of marks can be encoded as a list of (runtime) labeled natural numbers. Consider the lattice presented in Figure 5.10, that isolates two groups of students in two different classes. The program in Figure 5.11 unlabels the various labeled numbers it receives (❶), adds them up (❷) then returns their mean. Here, the contextual label resulting from a call to `mean` is useful for our computation: we would like to capture it, so that we know to whom the mean should be sent.

This section presents **DLIO**, an effect that unlike **GLIO**, keeps a runtime representation of current security label of computations. **DLIO** is a shallow layer above **GLIO**. A **DLIO** computation is represented directly as a **GLIO** computation; thus the IFC policy implemented by **DLIO** is exactly the one from **GLIO**. The effect **DLIO** does not bring supplementary trusted code base.

Note This section describes a new IFC effect; this section thus defines similar functions. We shadow certain definitions from Section 5.4; to refer to a shadowed definition `def` from that section, from now on, we write **GL**`.def`.

5.5.1 A Runtime Context

The type for contexts with runtime representations for current labels and clearance is defined below. The field `cle` is a computable predicate, while the field **GL**`.cle` (that is the field `cle` of the type `context` defined in Section 5.4) was for a non-computable predicate. Similarly, the `cur` field is a raw label, while the field **GL**`.cur` was erased. Relating inhabitants of

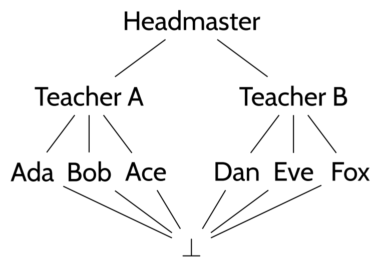

Fig. 5.10: Lattice representing a school hierarchy.

```
let mean (l:list (lv_rt ℕ)) =
  let marks = ❶ map unlabel l in
  let sum = ❷ fold (+) marks in
  sum / length l
```

Fig. 5.11: Example of a program dealing with labeled values of arbitrary labels.

context and GL.context is however easy; the relation ≡ is such that x ≡ y holds when the runtime-represented context x:context is and the erased context y:GL.context represents the same context.

```
noeq
type context = { cur: labelType
               ; cle: cle: (labelType → bool) {cle cur} }
let (≡) (c: context) (gc: GL.context): prop
  = Ghost.reveal gc.GL.cur == c.cur
  ∧ (∀ x. c.cle x ⟺ gc.GL.cle x)
```

The computations in the effect DLIO that this section aim at defining are equipped with a runtime-represented state: a context. We store the current state in memory, using Low* memory model through the minimal, IFC-aware, memory API we presented in Section 5.4.5. Consequently, the type pointer_context is a pointer to labeled contexts. The memory model of GLIO indeed enforces that pointers only reference labeled values. The type B.pointer is a refinement over B.buffer. A B.pointer τ is a pointer to a memory buffer of size one that holds a value of type τ.

```
type pointer_context = B.pointer (lv labelType context)
```

We omit the definition of two specifications: deref_context and as_ghost_context. Given m a specification model of a memory and pc a pointer to a context (of type pointer_context), deref_context m_0 pc is a computation in effect GTot that returns a context. Given c a runtime-represented context, as_ghost_context lifts c as a type-level context of type GL.context.

5.5.2 A Representation for DLIO

The specification monad for DLIO is straightforward: it reuses GLIO definitions, adding a concrete state. For instance, one refinement away, the type pre_t of pre-conditions for DLIO computations is defined as the arrow type pointer_context → GL.pre_t. As mentinoned earlier, DLIO is just a proxy of GLIO that mirrors type-level IFC operations into the world of computations, by keeping a runtime context around. Hence computations in DLIO are, without much surprise, simply represented as GLIO computations of the shape pointer_context→GLIO ⋯. The more rigorous definition for the representation of the effect DLIO is given below.

```
let repr (a: Type) (wp: wpₜ a) =
  pc:pointer_context
    → GLIO a
        (λ(p:GL.postₜ a) (c₀: GL.context) m₀ →
          wf_mem pc c₀ m₀ ∧ wp p pc c₀ m₀)
```

Note that wp_t refers to a DLIO variant of the type wp_t from GLIO. The predicate $\mathrm{wf}_{\mathrm{mem}}$ makes sure that the concrete context pointed at by pc reflects the erased **GLIO** context c_0. It also makes sure that pc points to a live area in memory. We skip the details about the effect definition itself: as the representation type points out, **DLIO** itself is almost just a regular state monad.

```
reifiable reflectable layered_effect {
  DLIO : a:Type → wp_t a → Effect
    with repr=repr; ⋯
}
```

We define the **DLio** Hoare-style effect variant of **DLIO**. Having our effect defined, let us now look at the IFC-related action `label`, `unlabel` and `toLabeled`, to fully understand how **DLIO** enjoys **GLIO** IFC policy enforcement.

5.5.3 Reflecting GLIO Computations

Lifting a **GLIO** computation as a **DLIO** computation is all about crafting a correct **DLIO** context. When a computation `f` of type `unit → GLIO τ ⋯` can be proven to leave its IFC context untouched, turning it into a **DLIO** computation is trivial. Consider `g = λ() _ → f ()`: it is a computation of type `unit →` *anything* `→ GLIO τ ⋯`. Fixing *anything* to `context`, `g` is actually of type `unit → repr τ ⋯`. Reflecting `g` thus gives a **DLIO** computation.

More concretely given a **GLIO** weakest-precondition `f`, `glio_wp_to_dlio f` computes its corresponding **DLIO** weakest-precondition. It alters `f` by injecting the systematic supplementary post-condition ❶. Recall $\texttt{wf}_{\texttt{mem}}$ `pc` c_1 m_1 spells out that the context pointed by `pc:pointer_context` at memory m_1 should be equivalent to the erased context c_1`:GL.context`. Consequently, even if a computation with weakest-precondition `f` seems to be free to alter its **GLIO** context, the computation at stake is also required to end its computation with a memory in which the runtime-represented context corresponds to the new context. In other words, if the computation does not update the memory pointed by pc, the computation cannot change its context.

```
let glio_wp_to_dlio (f: GL.wp_t τ): wp_t τ
  = λp pc c0 m0 →
      f (λ(x, c1) m1 → c0 ⋘ c1
                       ∧ ❶ wf_mem pc c1 m1
                       ∧ p (x, c1) m1
        ) c0 m0
```

Having this weakest-precondition mapping, the function $\texttt{run}^{\texttt{const}}_{\texttt{glio}}$ defined below, that maps **GLIO** computations to **DLIO** ones, is trivial to write.

```
let run_glio^const #a #wp ($f: unit → GLIO a wp)
  : DLIO a (glio_wp_to_dlio wp)
  = DLIO?.reflect (λ_ → f ())
```

However, this interface is not practical for computations which alter their IFC contexts. We instead define $\texttt{run}_{\texttt{glio}}$ `f c`, which runs a **GLIO** computation `f`, given a proof that `f` alters its context into exactly the context `c`.

```
let run_glio #a (c: context) #wp_f ($f: unit → GLIO a wp_f)
  : DLIO a (run^wp_glio wp_f c)
  = DLIO?.reflect (λpc → let r = f () in
                          GL.upd pc @ul c;
                          r
                 )
```

Notice the weakest-precondition involved: $\texttt{run}^{\texttt{wp}}_{\texttt{glio}}$ $\texttt{wp}_\texttt{f}$ `c`. Again, we need to transform **GLIO** weakest-preconditions into **DLIO** ones, but in a different way. Just as `glio_wp_to_dlio`, here, we add systematic post-conditions to the weakest-precondition $\texttt{wp}_\texttt{f}$. ❶ adds the requirement that the outcome context of the computation to be executed (modeled by the weakest-precondition $\texttt{wp}_\texttt{f}$ here) corresponds to the erased context $\texttt{c}_1$. After running a computation `f`, `run_glio_r` updates the pointer to the current context, whence the quantifier $\texttt{m}_2$. For every memory $\texttt{m}_2$ (❷) which is point-wise equal to memory $\texttt{m}_1$ except for address `pc` (❸), if the pointed context is well-formed in $\texttt{m}_2$ (❻) and is a labeled value protected by `c.cur` (❹) of value `c` (❺), then the post-condition at stake `p` should hold at memory $\texttt{m}_2$ (❼).

```
let run^wp_glio (wp_f: GL.wp_t τ) (c: context): wp_t τ
  = λp pc c0 m0 →
      wp_f (λ(x, c1) m1 →
           c0 ⋘ c1
         ∧ ❶ c ≡ c1
         ∧ B.live m1 pc
         ∧ (∀ ❷ m2. (❸ M.modifies (M.loc_buffer pc) m1 m2
                    ∧ ❹ deref_context m2 pc == c
                    ∧ ❺ label_of_context m2 pc == c.cur
                    ∧ ❻ wf_mem pc c1 m2 )
                    ⟹❼ p (x, c1) m2
           )
         ) c0 m0
```

Note that the computation to be run by both $\texttt{run}_{\texttt{glio}}$ and $\texttt{run}^{\texttt{const}}_{\texttt{glio}}$ is expected to be of type **unit**→**GLIO** ⋯, not of type **GLIO** ⋯. A computation in F*, disregarding the effect it is attached to, shall be an arrow type. Every other value is considered as a constant; a constant cannot be effectful, thus the type **E** ⋯ on its own (with **E** an effect) is forbidden. For convenience, the function `apply` turns any **GLIO** computation τ→**GLIO** β wp into τ→ **unit** →**GLIO** β wp.

```
let apply #a (#b: a → Type) (#wp: (i:a → GL.wp_t (b i)))
                             ($f:  (i:a → GLIO   (b i) (wp i)))
  : i:a → unit → GLIO (b i) (wp i)
  = λi _ → f i
```

The definitions apply, $\texttt{run}_{\texttt{glio}}$ and its variant $\texttt{run}^{\texttt{const}}_{\texttt{glio}}$ pave the road for bringing the IFC-related action of **GLIO** to **DLIO**.

5.5.4 Reflecting GLIO Actions

IFC-related actions mostly deal with labeled values. Our library provides two kinds of labeled values: we first bridge **GLIO** primitives on labeled values without runtime representation (indexed type lv), then we define wrappers for runtime-represented ones (indexed type $\texttt{lv}_{\texttt{rt}}$). Turning a value into an erased labeled value has no impact on the IFC context; thus the helper $\texttt{run}^{\texttt{const}}_{\texttt{glio}}$ is enough to bridge the action label. $\texttt{label}_{\texttt{rt}}$ labels values in a runtime labeled values.

```
let label #a (l: Ghost.erased labelType) (v: a)
  : DLIO (lv labelType a) (λp pc → label_wp l v p)
  = run_glio^const (apply (GL.label l) v)
let label_rt #a (l: labelType) (v: a)
  : DLIO (lv_rt labelType a)
       (λp pc c m → c.GL.cur ⊑ l
                 ∧ p ({v_rt = ghost_make l v; lbl_rt = l}, c) m)
  = let lv = label l v in
    assert (Ghost.reveal (labelOf lv) == l);
    { lbl_rt = l; v_rt = lv}
```

Unlabeling a piece of information possibly causes the raise of the current label; thus $\texttt{run}^{\texttt{const}}_{\texttt{glio}}$ is unsuitable. Below, unlabel uses $\texttt{run}_{\texttt{glio}}$ to bridge **GL**.unlabel, and replicates the type-level $\sqsubseteq$ performed by **GL**.unlabel in the world of computations. $\texttt{get}_{\texttt{ctx}}$ dereferences and unlabels the current (**DLIO**) context.

```
let unlabel #a (cur: labelType) (v: lv labelType a)
  : DLIO a (unlabel_wp cur v)
  = let c0 = get_ctx () in
    run_glio ({cur = c0.cur ⊔ cur; cle = c0.cle})
             (apply GL.unlabel v)
let unlabel_rt #a (runtime_v: lv_rt labelType a)
 : DLIO a (unlabel_wp runtime_v.lbl_rt runtime_v.v_rt )
  = unlabel runtime_v.lbl_rt runtime_v.v_rt
```

The last action we will take a look at will also be the most interesting one: toLabeled. Below we define its first variant, toLabeled_glio, a **DLIO** computation that runs a **GLIO** computation and wraps the raise of its contextual label as a labeled value. The purpose of the action **GL**.toLabeled is to capture any change to the current IFC context; as a result the IFC context of the computation **GL**.toLabeled f (for any f:**GLIO** ⋯) remains untouched. The action toLabeled_glio consequently bridges **GL**.toLabeled in a straightforward manner using $\texttt{run}^{\texttt{const}}_{\texttt{glio}}$.

```
let toLabeled_glio #a #wp_f ($f: unit → GLIO a wp_f)
  : DLIO (lv labelType a)
         (glio_wp_to_dlio (GL.toLabeled_wp  wp_f))
  = run^const_glio (λ_ → GL.toLabeled f)
```

Performing the operation toLabeled f where f is a **DLIO** computation requires one more step: we need to *reify* f into a **GLIO** computation, and then use the previously defined toLabeled_glio. Computation dlio_wp_to_glio is the opposite of glio_wp_to_dlio: given a pointer to a runtime context, it transforms a **DLIO** weakest-precondition into a **GLIO** one.

```
let dlio_wp_to_glio (d_wp : wp_t τ) pc: GL.wp_t τ
  = λp c_0 m_0 → wf_mem pc c_0 m_0 ∧ d_wp  p pc c_0 m_0
let toLabeled #a #wp_f (l: labelType) ($f: unit → DLIO a wp_f)
  : DLIO (lv labelType a) (λp pc c_0 m_0 →
        run^wp_glio (GL.toLabeled_wp (dlio_wp_to_glio wp_f pc))
                    (deref_context m_0 pc)
                    p pc c_0 m_0
  )
  = let pc = get_ctx_pointer () in
    toLabeled_glio' (λ_ → reify (f ()) pc)
```

5.6 An Example of Computation Mixing Statically and Dynamically Checked IFC Policy

In this section, we develop the school example introduced at the beginning of Section 5.5. We begin by giving an F* definition to the lattice informally introduced in Figure 5.10 through type labelType below. The value labelTypeLat implements an instance of the typeclass lattice for labelType which is presented in Figure 5.12. The lattice represents the organization of a school with two groups of students, group **A** and group **B**, which have each one teacher and a few students.

```
type group     = | A | B
type labelType = | Headmaster | Bot
                 | Teacher: group → labelType
                 | Student: string → group → labelType
instance labelTypeLat: lattice labelType = {
  ⊑ = (λx y → match x, y with
             | Bot, _ | _, Headmaster → true
             | Student _ gx, Teacher gy → gx = gy
             | _ → x = y);
  ⊔, refl_ord, trans_ord, antisym_ord, join_lub = ⋯; }
```

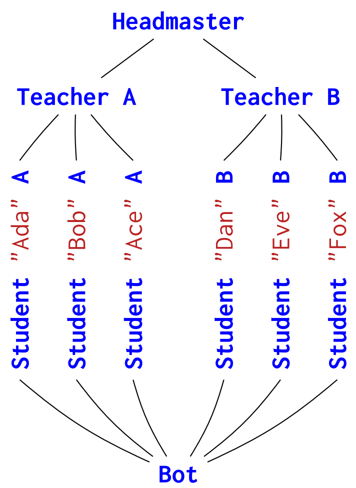

Fig. 5.12: Lattice representing a school hierarchy.

The IFC policy at stake in this example is that (i) no personal information from a student should fall into the hand of another student, and (ii) an information concerning a group of students should never be shared with another group of students. We consider the scenario where we send a report containing the mean of a set of marks (from –possibly– various students) to one person in the school (that is, a student, a teacher, or the headmaster). In this scenario, we don't want the mean value of the marks from group **A** to be sent to, e.g., a student in group **B** and vice-versa.

Let us consider the function `select_marks_of_group` below, for which we only give a type signature. Given a group g, it selects the marks of all the students that belong to group g. At ❶, the type signature of this function ensures that the maximally labeled value from the computed list `l` is at most labeled at **Teacher** g. This function does not alter the IFC context, as specified at ❷.

```
val select_marks_of_group: g: group
 → GLio (l: list (lv_rt labelType U32.t) {Cons? l})
        (λ_ _ → ⊤)
        (λc0 _ l c1 _ → ❷ c0 == c1 ∧ ❶ max Bot l ⊑ Teacher g)
let max cur (l: list (lv_rt labelType U32.t))
  = fold_left (⊔) cur (map (λx → x.lbl_rt ) l)
```

A list of marks emanating from `select_marks_of_group` is easy to deal with in **GLio**: the list provably contains only values labeled below a certain element of the lattice. The computation $\mathtt{ex}_1$ below computes a mean using this fact. In this scenario, there is no clearance policy: every computation is allowed to raise its current label to any degree. Encoding this absence of clearance leads to a certain verbosity, hence for the sake of clarity, we write the symbol ⓒ where we omit clearance-related predicates. The list `l` at ❶ is known to contain value labeled at most at **Teacher** g. The function $\mathtt{sum}_{\mathtt{glio}}$ is specified (at ❷) to raise the current label to the union of the maximum label contained in `l` and the current label. Since the initial current label of $\mathtt{ex}_1$ is specified (at ❸) to be **Bot**, it is trivial to prove statically that $\mathtt{sum}_{\mathtt{glio}}$ `l` will bring the current label to –at most– **Teacher** g. A report –which simply consists in the mean– is then sent, by the mean of the function `send_mean_to`, whose specification checks (at ❺) the current label to be lower than the label corresponding to the person the report is sent to.

```
let rec sum_glio (l: list (lv_rt labelType U32.t))
  : GLio U32.t (λc0 _ → ❸)
               (λc0 _ _ c1 _ → ❷ c1.cur == max c0.cur l ∧ ❸)
  = match l with
  | [] → 0ul
  | hd::tl → let hd = GL.unlabel hd.v_rt in
          hd + sum_glio tl
val send_mean_to: v:U32.t → who:labelType
  → GLio unit (λc0 m0 → ❹ c0.cur ⊑ who ∧ ❸) ⋯
let ex1 g (): GLio unit (λc0 _ → ❸ c0.cur == Bot ∧ ❸)
                        (λ⋯ → ⊤)
   = let l = ❶ select_marks_of_group g in
     let mean = sum_glio l / length_u32 l in
     send_mean_to mean (Teacher g)
```

Another scenario is a report about a set of marks of arbitrary students, embodied by definition `sample_marks`. We have no static knowledge about `sample_marks`: it might contain marks from only one specific student, a whole class or the whole school, we don't know. Computation $\mathtt{ex}_2$ is a **`DLio`** computation: it sums up the marks contained in `sample_marks`. Since we have no static knowledge about `sample_marks`, function $\mathtt{sum}_{\mathtt{dlio}}$ might raise the current label to any point in the lattice. Effect **`DLio`** keeping a representation of the current label, we can just use the current label to know to whom the report should be sent (that is the label `target` at ❶). Then, at ❷ we simply run the **`GLio`** computation `send_mean_to` to send the mean.

```
val sample_marks: l: list (lv_rt labelType U32.t) {Cons? l}
let rec sum_dlio (l: list (lv_rt labelType U32.t))
  : DLio U32.t  ⋯ // trivial pre/post
  = match l with | [] → 0ul
                 | hd::tl → let hd = unlabel_rt hd in
                         hd + sum_dlio tl
let ex2 (): DLio unit (λ_ c0 _ → c0.cur == Bot ∧ ❸) (λ⋯ → ⊤)
   = let mean = sum_dlio sample_marks
                / length_u32 sample_marks in
     let target = ❶ (DL.get_ctx ()).DL.cur in
     ❷ run_glio^const (apply (send_mean_to mean) target)
```

Below, computation `main` reuses $\mathtt{ex}_1$ and $\mathtt{ex}_2$ to compute three different means in two different ways: `main` combines computations in **`GLio`** (at ❶ and ❷) with a computation in **`DLio`** (at ❸). Each mean computation (at ❶, ❷ or ❸) inevitably raises the current label. To be able to compute three different means and send three different reports, we wrap them into `toLabeled`, and we discard their result.

```
let main (): GLio unit (λc₀ _ → c₀.cur == Bot ∧ ❸) (λ··· → ⊤)
 = let ctx = {DL.cur = Bot; DL.cle = (λ_ → true)} in
   let _ = ❸ toLabeled (apply (run_dlio_in_glio ex₂) ctx) in
   let _ = ❶ toLabeled (ex₁ A) in
   let _ = ❷ toLabeled (ex₁ B) in
   ()
```

5.7 A Tentative of Noninterference Proof Using Meta-Programming

In this section, we present our attempt to generate theorems of noninterference [Den76] using the meta-programming facilities offered by F⋆ (Meta-F⋆ [Mar+19]).

Noninterference of actual clients. The IFC system we implement in this chapter is a library on top of F⋆, and it uses the effect system of F⋆ to formulate flow control policies. Unlike some other approaches that consist in designing from the ground up a programming IFC-aware language, "IFC as a library" approaches [Ste+11; RCH09; Rus15; BVR15; Ste+17; PVH19] leverage their host language facilities (often Haskell [Pey07], Agda [CC99], and in our case F⋆) to encode an IFC system. One downside of such approaches is that proving a general property on the library at stake is complicated due to the host language. For instance, to prove that Haskell programs using a given library behave in a certain way, one would need to universally quantify the proof over all Haskell programs; such a proof is intractable. The common technique to overcome this intractability is to prove the desired property not on the library itself, but on a model of it. The resulting theorem about the desired property on the actual library consequently supposes one hypothesis: the model of the library and its implementation in the host language should have the same semantics.

The intractable aspect of a direct proof about the host language semantics is mainly due to the intractability of the host language semantics itself. Instead of considering a proof about any program under our library, our idea is to generate a mechanized and automated proof per library client program. In this way, a client can enjoy a noninterference proof generated in an ad-hoc manner, directly on the concrete library semantics, and not on a model.

Parametricity. Recently, parametricity [Rey83] has been applied to prove noninterference of clients of such "IFC as a library" approaches, directly on the library implementations. Parametricity has been successfully applied for both static [AB19] and dynamic [ABH21] IFC libraries. Such proofs emanate from a clever type encoding; consequently, they are concise and concern the library itself, not a model.

Parametricity however relies on Dependency Core Calculus [Aba+99], which does not handle side-effects. Our library is defined on top of the primitive effect **STATE** of Low*: a primitive effect is only about specifications (See 2.3.1), and provides no computational model, that is, no monad of computation. The side-effects of the clients of our effect **GLIO** thus cannot be represented in a monadic form. As a consequence, in our settings, applying parametricity is, at least, very challenging.

Meta-programming. Instead, we choose to leverage the meta-programming facilities F* provides means to implement a proof of noninterference based on erasure [LZ10; RCH08]. In this chapter, we need to be able to reify and normalize the IFC computations at stake; thus in throughout this chapter, by **GLIO** we denotes a variant of our effect **GLIO** that is not equipped with a memory API and whose representation is reifiable as a **GHOST** computation.

5.7.1 Noninterference: an Overview

We express the notion of noninterference of **GLIO** computations using a notion of `l`-view, with `l` being a label.

The `l`-view of a piece of information `i` is the visible information from the point of view of an observer allowed to see up to labeled information `l`. We derive the `l`-view of a value via an *erasure function*: every piece of information tainted with a label greater than `l` is replaced by a "hole" value •. As an example, consider the lattice **L** $\sqsubseteq$ **M** $\sqsubseteq$ **H**; the **L**-view of the list[5] `[⟨L,4⟩;⟨M,8⟩;⟨L,2⟩;⟨H,4⟩]` would be `[⟨L,4⟩;•;⟨L,2⟩;•]`. The syntax `⟨l,v⟩` denotes the value `v` labeled at `l`. The `l`-view of a value `v` of any type is given by `erase l v`.

[5] Note that here, the list itself is not labeled; only its elements are. While the elements of the list are protected, the list itself is not.

5.7.1.1 Standard encoding of noninterference.

Consider a computation `f` of type $\tau \to$ **GLIO** β $\mathtt{wp_f}$, with τ and β two types and $\mathtt{wp_f}$: **GL**.$\mathtt{wp_t}$ τ a weakest-precondition. The noninterference of `f` is commonly [PVH19] stated as in Equation 5.1. It states that the evaluation of `f` (i.e. $\downarrow^c\ (f\ x)$) and the evaluation of its erasure (i.e. $\downarrow^c\ ((\epsilon_l\ f)\ x)$) cannot be distinguished after erasure, for any erasure level l, input x and initial context c.

$$\forall l\ (x{:}\tau)\ (c{:}\mathtt{context}).\ \epsilon_l\ (\downarrow^c\ (\mathtt{f}\ x)) =_t \epsilon_l\ (\downarrow^c\ ((\epsilon_l\ \mathtt{f})\ x)) \qquad (5.1)$$

Note that this definition of noninterference treats function `f` as a term; whence the equality $=_t$ on terms. Similarly, function $\downarrow^c$ evaluates a term and function ϵ_l erases a term.

5.7.1.2 Encoding in F*.

In our settings, `f` is not a term but a computation. Consequently, this section reformulates the statement of Equation 5.1 accordingly.

Encoding of evaluation. For a given x: τ, f x is not a function, but a **GLIO** computation with a potential side effect; as such, one cannot evaluate this expression. Reification (See Figure 5.7 and Section 2.3.1) transforms f into its representation. The evaluation of $\downarrow^c$ f x is thus encoded as the expression **reify** (f x) c, where **reify** (f x) is of type **GL**.repr τ, a map from contexts to tuples of type $\tau \times$context.

Encoding of erasure. In Equation 5.1, the erasure is used to erase both (i) the results produced after evaluation and (ii) the function f itself. For the first case we define the eraseCtx function below.

```
class hasEraser τ
  = { erase : labelType → τ → GTot τ }
let eraseCtx {| hasEraser τ |} l ((x,c):(τ×context))
  : GTot τ
  = if c.cur ⊑ l then erase x else •
```

The function eraseCtx is designed to erase the tuple returned by **GLIO** representation. Its first argument is the label to erase at, and the second is the tuple produced after reifying a **GLIO** computation. When the label c.cur of the reified context is below the erasure label, the information x is observable from l. As illustrated in Figure 5.13, the value x might hold labeled values, thus we return erase l x.

Erasure on arrow type inhabitants is defined differently; the terms that constitute computations are erased separately by a meta-program. Our meta-program takes a top-level definition (say p of arrow type t), inspects its definition, and essentially produces an erased top-level p_erased of type labelType → t. More details about this meta-program are given in Section 5.7.3.

Noninterference Lemma Generation. The meta-program genNIStatement takes a name of an existing top-level **GLIO** computation, and generates a corresponding statement of noninterference.

```
let genNIStatement: name → Tac unit = ⋯
```

Unsurprisingly, genNIStatement is a **Tac** computation (See 2.3.2). The effect **Tac** offers an **API** for F* term inspection. The generation of the noninterference lemma is a side effect of genNIStatement; the function itself returns nothing, whence **unit**.

The **Tac** computations we saw (in lemma 2.7 or $\texttt{lemma}_{\texttt{inv}}$ of Section 3.5) were manipulating and solving proof goals. This is not the only use of **Tac** computations; as mentioned earlier, genNIStatement generates noninterference statements in the form of a new top-level definition. While the expression **assert** fact **by** tac invokes the tactic tac that solves the proof goal fact, the *declaration* %splice[$\texttt{tl}_0$;$\texttt{tl}_1$;⋯;$\texttt{tl}_n$] tac invokes the tactic tac that generates (at least) the top-levels named $\texttt{tl}_0$, $\texttt{tl}_1$, ... and $\texttt{tl}_n$. For simplicity we omit the top-level names and write %splice[⋯] tac.

```
( {cur = L; cle = ⋯}
, [ {lbl = H; v = 1;}
  ; {lbl = M; v = 2;}
  ; {lbl = L; v = 3;} ]
)

          |
      eraseCtx L
          ↓

  [ •
  ; •
  ; {lbl = L; v = 3;} ]
```

Fig. 5.13: Erasure of a tuple emanating from the representation of a **GLIO** computation.

```
let f (lv labelType τ): GLIO β ⋯ = ⋯
%splice[⋯](genNIStatement "f") // generates the lemma below
let f_ni = l:labelType → x:τ → c:context
  → Lemma (  eraseCtx l (reify (f x) c)
          == eraseCtx l (reify (f_erased l x) c))
```

The f_erased top-level declaration mentioned in $\mathtt{f}_{\mathtt{ni}}$ is also generated by the invocation genNIStatement "f".

5.7.2 Erasure of Values

Erasure of values is achieved by the method erase from the typeclass hasEraser. For a type τ that implements hasEraser and a value x:τ, erase l x is the value x where every piece of data inside x that is protected by a label higher than l is replaced with a “hole”. This hole is encoded in F* as •, an axiomatized polymorphic value, defined below; it has no content and can be used to erase any value.

```
assume val • (a: Type {∃ (x:a). ⊤}) → a
```

Erasing labeled values. Erasing labeled value at label l is the most interesting case: when its label is above l, we replace its content by a hole. Otherwise, it erases the data recursively on the structure of the type of the data (using typeclass inference mechanism). In both cases, the label remains untouched. Since we consider a variant of **GLIO** for which the representation is **GTot**, note that we are free to make decisions on erased labels and to construct labeled values (with $\mathtt{ghost}_{\mathtt{make}}$). We finally define lv_eraser an instance of the typeclass hasEraser for labeled τ.

```
let eraseLV {|hasEraser τ|} (l:labelType) (x:lv labelType τ)
  : GTot (lv τ)
  = ghost_make (labelOf x)
               (if labelOf x ⊑ l then erase l (valueOf x)
                                 else •)
instance lv_eraser {|hasEraser τ|}
  : hasEraser (labeled τ) = {
    erase = eraseLV
  }
```

Erasure of inductive inhabitants. For every primitive type (i.e. **unit**, **bool** or $\mathbb{Z}$) an eraser is just λ_ → id), where id is the identity. For inductive data types, we define a meta-program that can derive an instance of hasEraser inspecting its definition. For example, for lists, our meta-program follows the list constructors to mechanically generate the eraseList function below.

```
let rec eraseList {|hasEraser τ|} (l:labelType) (x:list τ)
  : GTot (list τ)
  = match x with
  | hd::tl → erase l hd :: eraseList l tl
  | [] → []
```

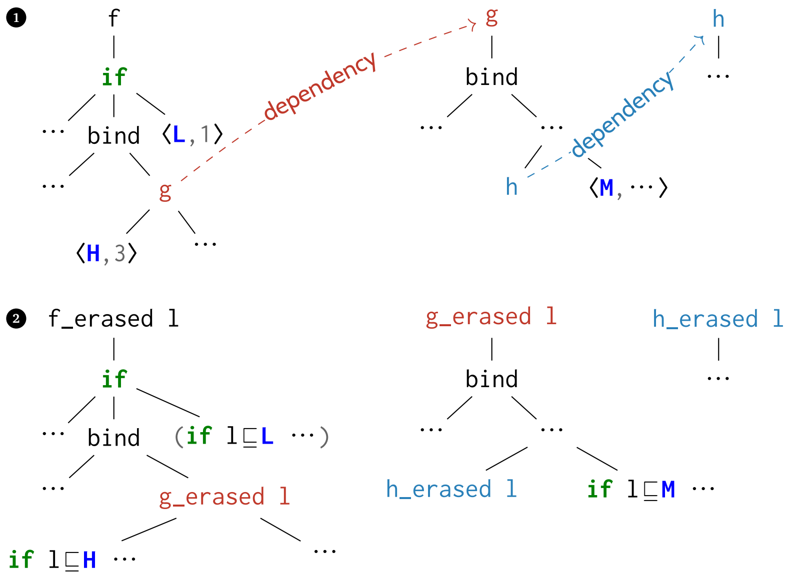

Fig. 5.14: Example of erasure for a top-level definition `f`. An example abstract syntactic tree for `f` is given at ❶, along with that of two other top-level definitions, `g` and `h`. Indeed, `f` depends on `h` and `g`; those two should be erased as well. ❷ presents the three generated top-level definitions.

5.7.3 Erasure of Computations

The meta-program `eraseF` erases computations. It is of type `name` → `Tac unit`: given the name of a top-level definition, it generates a number of erased top-level definitions. Figure 5.14 illustrates this process.

Consider a top-level "f"; `eraseF "f"` generates (at least) `f_erased`. If `f` has type `t`, then `f_erased` has type `labelType` → `t`: this first argument allows to set the erasure level. The body of the computation `f_erased` follows the AST of `f`, where (i) each labeled sub-term is erased and, (ii) each free variable referring to a top-level definition is replaced by its erased version, as explained below.

(i). Labeled Sub-term Erasure. For every sub-term `e` in the AST of `f`, when `e` represents a labeled value, it is replaced by `eraseLabeled` $\mathtt{l_e}$ `e`, with $\mathtt{l_e}$ the erasure level argument introduced in `f_erased`. To discriminate whether `e` is an erased label, we typecheck the AST of `f`[6] substituting `e` with the term `eraseLabeled` $\mathtt{l_e}$ `"e"`.

(ii). Erasure of Free-Variables. If `g` $\mathtt{v_0}$ ⋯ $\mathtt{v_1}$ is a sub-term of the AST of `f`, with `g` being a free-variable, then it means that `g` denotes a top-level definition. This also implies that the top-level `g` is a dependency for `f`. Thus, we replace this sub-term with `g_erased` $\mathtt{l_e}$ $\mathtt{v_0}$ ⋯ $\mathtt{v_1}$, $\mathtt{l_e}$ being the extra-parameter of `f_erased` for erasure level. We also collect the transitive closure of all the dependencies from `f` to other top-level definitions, and recursively generate erased version of each of them.

However there are some exceptions for such replacements of an external top-level `g`. When `g` refers to a definition that cannot be erased (i.e. only it's signature is known), and `g` is a constructor or refers to a type, then applying our computation erasure makes no sense, and in this case we leave the sub-term intact.

[6] At the time of writing, Meta-F*'s reflection API did not allow to easily alter the environment in which a sub-term is typed. Thus, we could not collect the unbounded free variables for a given sub-term and keep an environment of them for later typechecking. Hence, instead we typecheck an alteration of the AST of `f`.

5.7.4 Axiomatization of Contamination

Some operations in F⋆ are built-in, and enjoy no F⋆ implementation. For instance, the decidable equality = is built-in. Whether x = y for some x , y : τ (with τ a type equipped with decidable equality) is either decided by the SMT solver or by the normalizer of F⋆ (i.e. is the normal form of terms x and y equal?). Then, it is clear that neither the SMT solver nor the normalizer know what to decide about the equality • = 42. For the normalizer, • = 42 is stuck, i.e. no more reducing can be performed. There is no lemmas or facts that can either help the SMT solver to decide whether • = 42 is true or not.

This highlights a problem we call contamination. Contamination captures the propagation of • from arguments to results. Deciding whether • = 42 is true would require to observe what integer is •. The hole • absorbs this comparison, • = 42 shall be reduced to • itself[7]. The definition $\mathtt{contaminationDEq}_1$ axiomatizes the contamination of decidable equality in its first argument.

[7]Recall that • is an implicit polymorphic value. By revealing its implicit argument, we get that • #ℤ = 42 shall be reduced to • #bool

```
assume val contaminationDEq₁ x
  : Lemma ((• = x) == •)[SMTPat (• = x)]
```

In general if a function g consumes its ith argument, then the call of g with a • on the ith position should be reduced to •. Also, an inductive type that has only one constructor for which only one argument is informative, then it shall be erased as well for this argument.

5.7.5 Limitations

Our hope was that the computation-wise generated lemmas of noninterference would be, in general, simple enough to be proved automatically by the SMT solver. Unfortunately, this is the case only for trivial computations. Even simple non-recursive computations generate too complicated lemmas. One of the reasons for this struggling is our notion of contamination. Indeed, the notion adds a rule for normalization; we tried to integrate this rule as SMT patterns and with ad-hoc hand-written normalization processes, but we still hit some difficulties where some terms simply do not reduce as expected.

Also, our aim was to generate per-client proofs on their actual implementation. This aim is not fulfilled. Indeed, the representations of the effects of our library are Low⋆ computations: those are purely specificational, thus not reifiable. Our per-client lemmas are therefore stated against a lighter effect whose representation is **GTot**.

In the end, our meta-programming approach was non-trivial to implement, but we did not succeed to scale our approach to a whole soundness proof. All in all, we don't end up with a full and scalable proof of noninterference, but engineering this meta-program was however a quite enjoyable experience, that led to interesting developments. Computation erasure for instance led us to implement the $\mathtt{browse}_{\mathtt{term}}$ meta-program[8], which allows to browse, patch or collect information from an F⋆ term. Another example

[8]Which is available on GitHub: https://github.com/W95Psp/FStar-libs/blob/master/MetaTools/MetaTools.BrowseTerm.fst.

is the development of a meta-program that derives automatically a serializer and dezerializer given an inductive type definition; the Appendix A.1 gives more details and context about this meta-program.

5.8 Related Works

The IFC system presented in this chapter descends directly from Stefan's LIO [Ste+11], which itself descends from a vast line of works. It started with the basis of MAC [BL73] and with the general lattice-theoretical model proposed by Denning [Den76] to verify information flow policies.

Since then, a wide spectrum of systems has been described, from fully dynamic to fully static and from coarse to fine-grained systems. Our implementation, just as Stefan's LIO [Ste+11], is fine-grained, i.e. arbitrarily small pieces of data can be labeled.

From coarse-grained systems... Typically, IFC operating systems (e.g. [Efs+05]) are coarse grained: information flow policies are enforced and tracked at the level of processes or threads. As an example, the open-source RISC-V architecture supports extensions providing hardware IFC capabilities encoded as a byte-size tag alongside with data [Pal+18; Fer+18] to control data flow in accordance with the tag privileges. ARM's Trustzone allows to segregate encrypted and decrypted data in hardware-enforced trust zones. [De +15] generalizes this meta-data tag mechanism to implement more general software-defined IFC policies at hardware level. Virtualization technology and resource isolation available in modern operating systems and verified micro-kernels [Kle+09; Gu+16] is however far from available to consumer-market, IoT-oriented, embedded micro-controller architectures. On such targets, compartmentalization is a cost-effective compilation technique to complement label-enforced IFC policy with defensive code to isolate possible software faults and prevent program threads from addressing data outside of their designated partitions [De +15; Bes+19].

To fine-grained systems. Software-defined IFC helps to overcome hardware limitations and can, when available, strengthen coarse-grain, hardware security mechanisms (trust zones, virtualization, tags) with fine-grained user-, task- or channel-level micro-policies [De +15]. Software-level IFC was first proposed in [Mye99] to annotate Java programs with IFC policies. [HKS06] provides a language-agnostic library to check IFC properties in imperative C or Java programs.

Dynamic IFC policies have extensively been developed in operating system design. [Zel+06] provides a survey covering this domain. For instance, [Kro+07] proposes operating system mechanisms to systematically check information flow read or written by system threads.

IFC as a library. The concept of "IFC as a library", where the IFC system is hosted in another –expressive enough– language, was first proposed by

Li and Zdancewic [LZ06]. This work leverages arrows (a generalization of monads [Hug00]) to implement an IFC system in Haskell. Russo et al. [RCH09] shows that monads are enough to encode a library enforcing statically IFC in Haskell. The current state-of-the-art Haskell library was introduced by Stefan et al. [Ste+11], and followed by numerous LIO related works [Ste+12; BR13; BVR15; Ste+17; PVH19; GTA19]. Buiras et al. [BVR15] mixes static and dynamic verification in Haskell: they provide a `defer` primitive that captures certain kinds of static IFC constraints and defers them as runtime checks. Vazou et al. [PVH19] presents an extension to LIO that aims at transferring it usefulness to web applications: LWeb. LWeb provides a formalization of LIO extended with database transaction, along with a proof of non-interference using Liquid Haskell [VSJ14]. It also implements an extension to Yesod [Sno15], one of the main Haskell web frameworks. Gregersen et al. [GTA19] presents an IFC library, DepSec, inspired by MAC [Vas+18]. This library is implemented in Idris, a dependently typed language: DepSec investigates the extra expressiveness brought by such a type system.

In parallel, Austin et al. [AF12; ASF17; Yan+16] develop the idea of faceted values, i.e., tuples of dimension n holding the n different views for each of the n different security labels. This approach is highly dynamic.

The application of software-defined IFC policies to embedded devices with, e.g. LIO, faces two major obstacles. First, the policy enforcement of LIO relies on automatically generated runtime checks that could, if not properly sand-boxed, cause a device to crash unpredictably because of an IFC exception. Second, embedded systems have limited resources: the "IFC as a library" approach relying on facilities generally implemented by high-level and garbage collected languages (i.e. monads and strong type systems), such libraries are often not well-suited.

Our approach takes advantage of both the expressiveness of dependent-types in the verified programming language F* [Swa+16] allowing us to use F* effects to encode monadic IFC encapsulation, and the capability of generating possibly zero-runtime C system code, by using its KreMLin [Pro+17] code generator.

Like related approaches based on high-level programming languages, [Ste+11; GTA19; BVR15], our library offers a lot of flexibility in the IFC policy enforcement, and allows from runtime checks to static proof obligations by using its powerful type system.

[BVR15] offers a different hybridation mechanism than ours: it eliminates IFC runtime checks that can be ruled safe statically and keeps other, call-dependent, dynamic checks. This is a more appropriate approach for transactional applications, where throwing an exception from some LIO client application is non-critical or fail-safe. However, in the case of –possibly unattended– reactive applications, this is not an option, as failing safe usually means to restart a real-time and potentially mission- or safety-critical application.

[SR09] provides a detailed review on the extensive number of related approaches based on the static analysis of imperative system programs. The recent [Gua+20], for instance, statically analyses bytecode to monitor programs that may leak unintended information when executed on speculative architectures. As in these approaches, our library offers the capability to run verified code generated from the KreMLin compiler [Pro+17], without the need for a runtime library or a garbage collector, and hence for direct application for low-level, resource-constrained, embedded architectures.

5.9 Conclusion

This chapter presented an IFC framework designed with F⋆ and leveraging its effects system. Our library offers a compatibility –to some degree– with the Low⋆ subset of F⋆, since a client of our library (i) enjoys C extraction via KreMLin [Pro+17] when it is written in Low⋆, and (ii) is able to deal with a –small– fraction of the memory model of Low⋆.

The implementation of our library is split into two parts. The first one is **`GLIO`**, which provides a fully static IFC system. A client of **`GLIO`** enjoys zero runtime costs: the **`GLIO`** bits of our library have no runtime representation. However, a client of **`GLIO`** shall prove statically that it is respecting its IFC policy. Such static proofs can be time consuming, and especially can be redundant with runtime operations: when the IFC policy and the data coincide, checking –at least partially– IFC policies at runtime can be relevant. The second part of our library consists in effect **`DLIO`**, which precisely allows to verify an IFC policy in a more dynamical way. Static and dynamic IFC are complementary: our library allows the programmer to compose them together, according to the needs.

We also present a way of generating noninterference theorem statements via meta-programming; however, as discussed in Section 5.7.5, this approach suffers of some limitations. We miss an empirical evaluation of our library. Thus, as a future work, we would like to implement a motivating example for our library by implementing a real-word application verified to comply to an IFC policy using both effects **`GLIO`** and **`DLIO`**. We also aim at proving our library to ensure noninterference as a future work.

CHAPTER 6

Conclusion: Summary and Perspectives

6.1 Overview

An advanced type system –featuring dependent types for instance– offers a great degree of expressiveness, but at the cost of an additional human cost, in the form of manual annotations and proofs. The approach of static analysis helps at inferring automatically semantic properties about programs. In this manuscript, we studied several kinds of interactions between static analysis methods and advanced type systems.

First, we focused on the advantages of using a smart and strong type system for writing robust static analysis tools and proving their correctness. Following this idea, we presented a static analyzer which implements abstract interpretation algorithms and which enjoys a quite concise and understandable F* implementation. This analyzer targets a simple imperative language and implements the abstract domain of intervals, but enjoys a modular design and remains accessible to understand. By leveraging refinement types the components of our abstract interpretation are given strong types, that directly encode theorems of soundness in a very clear and intelligible way. Thanks to F* automation, the amount of manual proofs required in our implementation is an order of magnitude less in comparison with similar work.

Then, we looked at how such a verified abstract interpreter could, in turn, help F* type inference. The procedure for type-checking a fragment of code in F* consists in (i) building up a *proof obligation* via dedicated weakest-precondition calculi implemented by effects, and then (ii) relies on an SMT solver to discharge them automatically. Our idea was to operate directly at the effect level of F*, which allows for a great modularity in verification. An effect implements a weakest-precondition calculus: our approach consists in hybridizing verified abstract interpreters with weakest-precondition monads. In doing so, the weakest-precondition monads are enriched with abstract interpreter reasoning and its inference abilities, injecting invariants on-the-fly. In the end, this approach results in lighter proof obligations and less manual annotations for the F* programmer. To summarize, the work we have presented turns abstract interpreters into weakest-precondition monad transformers. It allows for interactions between weakest-precondition computations and static analyses. Our transformed hybrid monads however currently yield an exponential number of abstract analyses, which is a severe limitation for practical use.

Finally, we investigated how a specific kind of analysis (namely, Information Control Flow policies) could be encoded both statically and dynamically as F⋆ effects. We presented the implementation and design of a library that allows the verifications of IFC properties on F⋆ programs. The library offers both static and dynamic verification of IFC policies; a client is free to use any mix of static and dynamic verification, at her/his convenience. It was also designed with low-level and embedded software in mind. Indeed, the choice of a dynamic verification of a policy has consequences in terms of memory usage and runtime performance; instead, our library lets the user choose the right balance between runtime cost and proof efforts. In this perspective, our library is written in Low⋆ subset of F⋆, which enjoys an extraction procedure to C and WebAssembly code via the KreMLin tool. Consequently, a Low⋆ client of our library also enjoys this low-level code extraction. Finally, we also presented an attempt at proving the noninterference of the clients of our library. Instead of proving noninterference on a model of our library for any client, the idea was to leverage meta-programming to automatically generate theorems of noninterference per-client. This approach however turned out to yield a lot of complexity, both in terms of specification and for the SMT solver.

6.2 Perspectives

6.2.1 A Low-Level Verified Abstract Interpreter Implemented in Low⋆

The runtime efficiency of most verified abstract interpreters is poor, the emphasis being placed mostly on soundness. For instance, the static analyzer Verasco is equipped with advanced abstract interpretation features and targets an important subset of the C language, but takes a very long time to analyse programs [Jou+]. F⋆ has been very successful to implement verified low-level algorithms, using Low⋆, a subset of F⋆ for which the tool KreMLin provides an extraction process to C code. Low⋆ is a C DSL embedded in F⋆: while the Low⋆ code is low-level and resembles C, specifications and proofs still enjoy full F⋆ features. This opens the path for implementing a verified, low-level and efficient abstract interpreter in Low⋆. Low-level code and low-level data structure yield more complexity than their functional counterpart and therefore generate more complicated and verbose proofs. In addition to the benefit of having an efficient verified abstract interpreter, it would be interesting to observe the amount of proof effort it would require.

6.2.2 Implement and Connect More Powerful Abstract Domains

The abstract interpreter we presented in Chapter 3 is modular, and takes abstract domains as a parameter. We implemented only the abstract domain of intervals and we observed it did not require a great amount of manual proofs. This leads to questioning whether this lightness in terms of proof would carry over more complicated domains. For example, implementing Karr's Domain [Kar76] requires different kinds of algorithms and proofs (i.e. algorithms from basic algebra dealing with matrices, such as Gaussian elimination) in comparison with the domains of intervals for instance. Equipped with more domains, our abstract interpreter would also benefit from abstract domain transformers such as the product domain.

6.2.3 More Powerful Formalization of Memory Abstractions

The interface for memory abstract domains in our abstract interpreter reflects the expressiveness of our target language IMP. In consequence, it is quite weak, and doesn't support e.g. pointer arithmetic. A natural extension to our abstract interpreter is to gradually enrich our target language to support more advanced features, to eventually reach a real-world language.

6.2.4 Our Hybridization and its Exponential Analysis Time

Our hybridization methodology interleaves abstract interpretation with weakest-precondition calculus too closely. This tight encoding causes us problems to *compute* certain abstract states: when analyzing a conditional, we fork abstract analysis and weakest-precondition calculus in two, but we are not able to merge resulting analysis back. Consequently, analyzing a sequence of n conditionals results in 2^n independent abstract analysis.

An Impossibility? The first step into investigating further our idea of weakest-precondition monad hybridization through abstract interpretation would be to evaluate whether the premises to our hybridization lead to an impossibility for computing abstract joins. Stating and proving such an impossibility would be interesting but quite a theoretical challenge.

Free Monads to the Rescue! The difficulty in joining back forked abstract interpretation comes from the tight encoding of our hybridization. A simple way of loosening this encoding would be to make use of free monads [Swi08]. A monad usually "computes" something: binding two computations collapses their respective contexts. Free monads are monadic structure nesting their contexts, leaving their user choose later how to interpret and how to give meaning to this nesting. Our encoding requires to interleave abstract interpretations and weakest-precondition computations at every monadic *step*. Given a computation, our encoding produces a hybrid representation atomically, from which we cannot extract an abstract interpretation without yielding computing a weakest-precondition in the same time. Using a free monad, we could enjoy a convenient intermediate representation, designed so that we can independently derive an abstract interpretation or a weakest-precondition. Then, from this intermediate representation, we could design a procedure to output hybrid weakest-preconditions.

6.2.5 Real Effects

The problem concerning the impossibility of joining abstract interpretations put aside, an interesting extension to our hybridization method would be to scale it up to real F* effects. Our approach defines a *specification* monad transformer and an F* effect is usually a combination of computational and specification monads. *Layered effects* allow one to define effects whose representations are themselves computations living in other effects. Thus, layered effects seem to be the perfect fit for implementing hybrid effects: a hybrid effect would then be an effect layered onto the original effect to be hybridized, but with a different specification monad. Technically, writing a hybridized effect (for instance, the one of Section 4.4) as a layered effect should not be too difficult. Writing the transformer itself would require much more technical work however, as it would require writing a meta-program that inspects and generates layered effect definitions, which is not possible with Meta-F* at the time of writing.

6.2.6 Proving Noninterference with Parametricity

Our approach to noninterference in our IFC library has some flaws. Our idea was to generate noninterference statements per-client so as noninterference be proved on actual clients and library code, not on a model of our library. However, the meta-programming procedure that generates these statements is far from trivial. Thus, it is hard to be convinced that the generated statements coincide with noninterference. In other words, our meta-program is not verified to generate correct noninterference statements. Also, while we expected the proof efforts to be low for our generated statements, we observed that even for trivial cases we needed extensive manual proofs. Thus, our library would benefit from a radically different approach. Recent related works [ABH21; AB19] proceed in proving noninterference via *parametricity*, and are very promising. However, such proofs have only been demonstrated for pure IFC libraries, free of side-effects, even though some theoretical results have appeared on encoding parametricity in effectful contexts [AR17]. Such a proof via parametricity for our library would therefore yield both interesting technical and theoretical challenges.

6.2.7 Experimenting with F*

In this manuscript, most of the work was conducted using the F* programming language: working and hacking on F* was a quite pleasant experience. During the three years of my Ph.D. studies, I wrote about 50k[1] lines of F* code (excluding blanks or comments), of which about 2k are related to Chapter 3, 18k to Chapter 4, 16k to Chapter 5, the rest being modules or libraries shared among the chapters. Of course, not everything is rosy, and certain rough edges sometimes slowed us down. For instance, F* typeclasses are interesting to work with in F* since they are essentially an user-level feature, almost entirely implemented as Meta-F*. In consequence, it is easy to hack on typeclasses (i.e. extract or generate their definitions via meta-programming), but also its instance inference

[1] This number was obtained by analysing my Git repositories hosted on GitHub and on Inria's GitLab.

mechanism is not very powerful. For instance, it is possible to use F* typeclasses to implement type families [KJS10], but this almost breaks the inference. Designing the typeclasses of Chapter 3 was a bit of a balance act between good inference and clean abstraction.

The reflection facilities provided by Meta-F* are not complete yet; for instance, type universes or effect definitions are not exposed. Some transformations of the noninterference meta-program in Chapter 5 have thus been a bit complicated. Generation of effects via meta-programming would also help for the extension discussed in Section 6.2.5.

A great feature of F* is its extraction to C through the tool KreMLin. The distinction between which F* code is included in the Low* subset and which is not can be quite complicated to grasp. This distinction is ultimately decided by the OCaml implementation of KreMLin itself. Using meta-programming, normalization and other such optimizations, it is possible to extract astonishingly high-level and abstract F* code to C, but this process is tedious.

However, low-level programming in F* has a bright future: the F* community is already developing Steel [Swa+20] which is a very promising successor to Low*. Experimenting and writing verified software –for example a low-level sound abstract interpreter– with Steel is the way to make it more robust and more accessible.

APPENDIX A

A Selection of F⋆ Implementations

This manuscript presented three main works: a verified abstract interpreter (Chapter 3), a specification monad transformer (Chapter 4), and a framework for static to dynamic verification of IFC policies (Chapter 5). Each of them required a certain amount of F⋆ implementation; this appendix briefly presents a selection of some of the technical challenges encountered.

A.1 Marshaling, Native Execution and Meta-Programming

Running an F⋆ program consists in compiling and running its extracted OCaml (or e.g. C) code. One can also use the typechecker of F⋆ to normalize a term, and by this means, "run" total computations. However, running a meta-program only makes sense during the type-checking phase: a meta-program might for instance interact with a proof state. By default, meta-programs are run through normalization allowing the user to write meta-programs and use them in a flexible way on-the-fly. Normalization is however not very efficient: to tackle performance issues, F⋆ allows for *plugins*.

A top-level definition can be marked with the `plugin` attribute. A module with such `plugin` top-levels can then be extracted to OCaml and compiled as a native shared library. Then one can plug this native library back into F⋆ so as the compiled top-levels marked with `plugin` are executed with native performance instead of being normalized. Of course, substituting a normalization into a call to a native compiled function in the middle of a normalization process is not trivial. This involves some marshalling from *term representations* (that is abstract syntax trees) to *native representations*, and back. For instance, consider the following function `sum`, that computes the sum of all numbers:

```
[@@plugin]
let rec sum (l: list ℤ): ℤ
  = match l with
  | [] → 0 | hd::tl → hd + sum tl
```

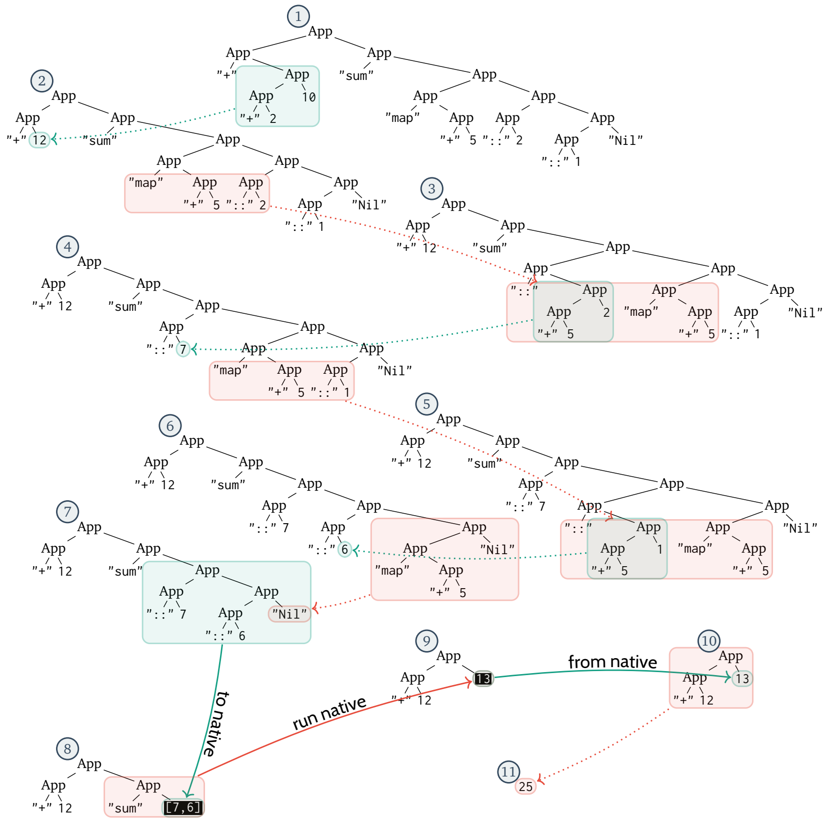

Fig. A.1: Illustration of the successive abstract syntactic trees for normalizing the term `2 + 10 + sum (map (+ 5) (2::1::Nil))`. Changes in the successive trees are highlighted with colors. Values with a black background are native representations; everything else being terms. The plain and thick arrows indicate type marshalling between native and term representations, or indicate native execution.

Figure A.1 illustrates the steps involved in the normalization of the term `2 + 10 + sum (map (+ 5) (2::1::Nil))`, when `sum` is native. The last steps (7–8, 8–9 and 9–10) are particularly interesting: to invoke the function `sum`, its argument should be completely normalized. Indeed, the normal form of a term of type `list` is (i) either the top-level name `"Nil"` (and then is represented as a native empty list), or (ii) a binary application of the top-level name `"::"` (the second constructor of the inductive `list`) to some list elements and some other lists. Once we have a native representation `r` (see tree 8), running the native program `sum r` returns a native result (here an integer, see tree 9), which shall be transformed back to a term.

F* defines such a transformation from terms to native representations (and back) only for a small set of builtin types, such as lists, options, integers, etc. F* provides no bridge between terms and native representation for used-defined inductive types: a plugin whose type signatures contain a type for which no bridge exists is rejected and cannot be extracted or compiled. For instance, the first implementation of the transformer presented in Chapter 4 was performing quite heavy transformations on custom inductive types. Computing the hybrid weakest-precondition of a simple program was taking a few minutes: compiling this procedure was thus necessary.

To circumvent this issue, we wrote a meta-program that automatically generates serializers and deserializers for a given inductive type. The representation for serialized data is a type for which F* can derive a native representation. At the cost of an indirection through serializers and deserializers, this process enables any function to be a plugin. Below is an example of extraction for a function `f` which involves two user-defined types `foo` and `bar`. At ❶ and ❷, we invoke the meta-program `generateSerialize`, which generates the appropriate serializing functions. Then, at ❸ the function f_{ser} defines an indirection through serialization and deserialization to the original function `f`. f_{ser} is marked as a plugin, and has a correct type for native compilation. Finally, at ❹ the function f_{natv} goes the other way around and restores the type signature of function `f`.

```
type foo = ···        type bar = ···
❶ %splice[···] (generateSerialize (`%foo));
❷ %splice[···] (generateSerialize (`%bar));
let f: foo → bar = ···
[@@plugin]
❸ let f_ser (x: serialized): serialized
    = serialize (f (deserialize x))
❹ let f_natv (x: foo): bar
    = deserialize (f (serialize x))
```

This development of this meta-program can be found at the following URL on GitHub:

`github.com/W95Psp/FStar-libs/tree/master/Data/Serialize`

A.2 Parser Combinators and Pretty Printers

In Chapter 3, we implement an abstract interpreter that analyses a small imperative language. The prototype reads and parses files of this small language; it also pretty prints abstract syntax trees. Instead of delegating these tasks to an OCaml module for instance, we implemented two small F⋆ libraries that provide parser combinators and pretty-printing facilities.

Parser combinators. A common approach to lexing and parsing consists in describing a grammar in a dialect of EBNF (Extended Backus-Naur Form), and then in using a parser generator to transform it into an actual executable parser. Parser combinators are a functional solution to parsing. They are higher-order functions transforming one or several input parsers into new parsers in output. Starting with basic building blocks (i.e. parsers that exactly match one character), it is easy to combine them into bigger parsers. For instance, given `digit: parser` $\mathbb{Z}$ a parser for digits, it is easy to parse numbers, using the $\texttt{many}_1$`: parser` $\tau \to$ `parser (l: list` τ `{Cons? l})` sequence combinator: $\texttt{many}_1$ `digit` is a parser for sequences of digits, of type `l:list` $\mathbb{Z}$ `{Cons? l}`. We implemented **`StarCombinator`**, a small parser combinator library which defines basic parsers and combinators. Its development can be found at the following URL on GitHub:

`https://github.com/W95Psp/StarCombinator/`

Pretty printer. For convenience, we also needed to be able to print the trees we were parsing. Decent pretty printing of an AST is not so simple to achieve; and abstracting away printing and formatting matters can greatly enhance and simplify pretty printing. Thus, in the spirit of Wadler [Wad03], we wrote a small module that provides a bunch of functions to help constructing documents (an inductive type that represents pretty-printed to be strings), which then can be transformed into strings. This module can be found at the following URL on GitHub:

`https://github.com/W95Psp/verified-abstract-interpreter`
`/blob/master/src/app/PrettyPrinter/`

A.3 Nix and F⋆

F⋆ is an ever changing language, and every so often there is a bug fix or a new functionality pushed on some branch of F⋆ repository. Also, we spend quite some time hacking on F⋆ itself to tweak it, or add small features[1]. As a result, we tend to work with a certain number of different versions of F⋆, often with custom patches.

The Nix Package Manger [NixPM] is a purely functional package manager, leveraging the Nix functional lazy programming language. It is

[1] For instance for tweaking the reflection API (i.e. adding range, comment, lemma, parsing reflection API), or playing with certain F⋆ internals.

focused on reproducibility: building a same package on two different computers (with same CPU architecture) must result in entirely bit-to-bit identical binaries. Nix builds software in a side-effect free manner. Each package is isolated and identified with a cryptographic hash of the build dependency graph of the package.

We wrote a Nix expression that allows easy compilation of any number of versions of F* given a set of options, patches and sources. The abstract interpreter of Chapter 3 leverages these Nix expressions, resulting in an easy-to-reproduce development environment and binaries. This Nix expression is available at the following URL on GitHub:

`https://github.com/W95Psp/nix-flake-fstar`

After using F* for a few years on very various projects, we experimented a slight inconvenience: the F* ecosystem did not develop any notion of third-party libraries. We often felt the urge of extracting certain modules from a project, to isolate them into a small library. But since there is no ready-made path to declare and reuse libraries, such F* libraries seem rare. In consequence, the only F* library is basically the standard one [FStdLib]. But clearly, every module does not belong to the standard library of a language. We thus wrote a small package manager for F* using Nix. It allows to describe GitHub-based or local dependencies for a library. Also, it generates development environments with pre-configured F* binaries and auto-generation of native plugins, if any. The package manager also lets the user to specify OCaml compilation targets. We aim at simplifying this package manager and at augmenting it with C and JavaScript compilation target. It is available at the following URL on GitHub:

`https://github.com/W95Psp/fstar-nix-packer`

List of Figures

Bibliography

[WR10] Alfred North Whitehead and Bertrand Russell. Principia mathematica; 1nd ed. Cambridge, UK, 1910.

[Chu36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. 1936. DOI: 10.2307/2268571.

[Tur37] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. 1937. DOI: 10.1112/plms/s2-42.1.230.

[BL73] David Elliott Bell and Leonard J. LaPadula. Secure Computer Systems: Mathematical Foundations. In: 1973.

[Den76] Dorothy E Denning. A Lattice Model of Secure Information Flow. 1976. DOI: 10.1145/360051.360056.

[Kar76] Michael Karr. Affine Relationships among Variables of a Program. 1976. DOI: 10.1007/BF00268497.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '77*. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[Rey83] John C Reynolds. Types, Abstraction and Parametric Polymorphism. In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 1983, pp. 513–523. DOI: 10.1007/3-540-55511-0_1.

[CH86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. INRIA, 1986.

[Mog91] Eugenio Moggi. Notions of Computation and Monads. 1991. DOI: 10.1016/0890-5401(91)90052-4.

[PW93] Simon Peyton Jones and Philip Wadler. Imperative Functional Programming. In: 1993, pp. 71–84. DOI: 10.1145/158511.158524.

[Aba+99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. A Core Calculus of Dependency. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1999, pp. 147–160. DOI: 10.1145/292540.292555.

[CC99] Catarina Coquand and Thierry Coquand. Structured Type Theory. In: *Workshop on Logical Frameworks and Metalanguages*. 1999.

[Mye99] Andrew C Myers. JFlow: Practical Mostly-Static Information Flow Control. In: *Proceedings of the 26th Symposium on Principles of Programming Languages (POPL'99)*. 1999, pp. 228–241. DOI: 10.1145/292540.292561.

[Hug00] John Hughes. Generalising Monads to Arrows. 2000. DOI: 10.1016/S0167-6423(99)00023-4.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Vol. 2283. 2002. URL: https://www21.in.tum.de/~nipkow/LNCS2283/.

[Pla02] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. 2002.

[Wad03] Philip Wadler. A prettier printer. 2003. DOI: 10.1007/978-1-349-91518-7_11.

[The04] The Coq development team. The Coq proof assistant reference manual. Version 8.0. 2004. URL: http://coq.inria.fr.

[Cou+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In: *Proceedings of the European Symposium on Programming (ESOP'05)*. 2005 LNCS, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[Dav05] Pichardie David. Interprétation Abstraite En Logique Intuitionniste : Extraction d'analyseurs Java Certifiés. Université Rennes 1, 2005. URL: https://davidpichardie.github.io/papers/these-pichardie.pdf.

[Efs+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. 2005. DOI: 10.1145/1095809.1095813.

[HKS06] Christian Hammer, Jens Krinke, and Gregor Snelting. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In: *IEEE International Symposium on Secure Software Engineering*. 2006, pp. 87–96.

[LZ06] Peng Li and Steve Zdancewic. Encoding Information Flow in Haskell. In: *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 2006, 12–pp. DOI: 10.1109/CSFW.2006.13.

[Zel+06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. 2006, pp. 263–278. DOI: 10.5555/1267308.1267327.

[Kro+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In: *ACM SIGOPS Operating Systems Review*. 2007 6, pp. 321–334. DOI: 10.1145/1323293.1294293.

[Pey07] Simon Peyton Jones. A History of Haskell: Being Lazy with Class. In: *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*. 2007. DOI: 10.1145/1238844.1238856.

[Ber08] Yves Bertot. Structural Abstract Interpretation, A Formal Study Using Coq. In: *LERNET Summer School*. 2008. DOI: 10.1007/978-3-642-03153-3_4.

[DB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MKJ08] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'08)*. 2008, pp. 159–169. DOI: 10.1145/1375581.1375602.

[NMB08] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. 2008. DOI: 10.1017/S0956796808006953.

[Nan+08] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent Types for Imperative Programs. In: *Proceedings of the 13th International Conference on Functional Programming (IFCP'13)*. 2008, pp. 229–240. DOI: 10.1145/1411203.1411237.

[RCH08] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. 2008.

[Swi08] Wouter Swierstra. Data Types à La Carte. 2008. DOI: 10.1017/S0956796808006758.

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal Verification of an OS Kernel. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[RCH09] Alejandro Russo, Koen Claessen, and John Hughes. A Library for Light-Weight Information-Flow Security in Haskell. 2009. DOI: 10.1145/1543134.1411289.

[SR09] Andrei Sabelfeld and Alejandro Russo. From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In: *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*. 2009 PSI'09, pp. 352–365. DOI: 10.1007/978-3-642-11486-1_30.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The Smt-Lib Standard: Version 2.0. In: *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, England)*. 2010, p. 14.

[CP10] David Cachera and David Pichardie. A Certified Denotational Abstract Interpreter. In: *Proc. of International Conference on Interactive Theorem Proving (ITP-10)*. 2010 Lecture Notes in Computer Science, pp. 9–24. DOI: 10.1007/978-3-642-14052-5_3.

[KJS10] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In: *Reflections on the Work of CAR Hoare*. 2010, pp. 301–331.

[LZ10] Peng Li and Steve Zdancewic. Arrows for Secure Information Flow. 2010. DOI: 10.1016/j.tcs.2010.01.025.

[Ste+11] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazieres. Flexible Dynamic Information Flow Control in Haskell. 2011.

[AF12] Thomas H. Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2012 POPL '12, pp. 165–178. DOI: 10.1145/2103656.2103677.

[Nip12] Tobias Nipkow. Abstract Interpretation of Annotated Commands. In: *Interactive Theorem Proving (ITP 2012)*. 2012 LNCS, pp. 116–132.

[Ste+12] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C Mitchell, and David Mazieres. Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems. 2012. DOI: 10.1145/2398856.2364557.

[Bra13] Edwin Brady. Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation. 2013. DOI: 10.1017/S095679681300018X.

[BR13] Pablo Buiras and Alejandro Russo. Lazy Programs Leak Secrets. In: *Nordic Conference on Secure IT Systems*. 2013, pp. 116–122. DOI: 10.1007/978-3-642-41488-6_8.

[Swa+13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In: *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*. 2013 PLDI '13, pp. 387–398. DOI: 10.1145/2491956.2491978.

[Koo14] Phil Koopman. A Case Study of Toyota Unintended Acceleration and Software Safety. 2014.

[VSJ14] Niki Vazou, Eric L Seidel, and Ranjit Jhala. Liquidhaskell: Experience with Refinement Types in the Real World. In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. 2014, pp. 39–51. DOI: 10.1145/2775050.2633366.

[Vaz+14] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. 2014, pp. 269–282. DOI: 10.1145/2628136.2628161.

[BVR15] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In: *ACM SIGPLAN Notices*. 2015 9, pp. 289–301. DOI: 10.1145/2784731.2784758.

[DMV15] David Darais, Matthew Might, and David Van Horn. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2015, pp. 552–571. DOI: 10.1145/2814270.2814308.

[De +15] Arthur Azevedo De Amorim, Maxime Dénes, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 813–830. DOI: 10.1109/SP.2015.55.

[Lap15] Vincent Laporte. Verified Static Analyzes for Low-Level Languages. Theses. Université Rennes 1, Nov. 2015. URL: https://tel.archives-ouvertes.fr/tel-01285624.

[Mou+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In: *Automated Deduction - CADE-25, 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 2015.

[Rus15] Alejandro Russo. Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell. 2015. DOI: 10.1145/2784731.2784756.

[Sno15] Michael Snoyman. Developing Web Apps with Haskell and Yesod: Safety-Driven Web Development. 2015. ISBN: 9781491915592.

[BLP16] Sandrine Blazy, Vincent Laporte, and David Pichardie. An Abstract Memory Functor for Verified C Static Analyzers. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016*. 2016, pp. 325–337. DOI: 10.1145/2951913.2951937.

[Gu+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent {OS} Kernels. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 653–669. DOI: CertiKOS : AnExtensibleArchitectureforBuildingCertifiedConcurrent.

[Jou16] Jacques-Henri Jourdan. Verasco: A Formally Verified c Static Analyzer. Theses. Universite Paris Diderot-Paris VII, May 2016. URL: https://hal.archives-ouvertes.fr/tel-01327023.

[Ler+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – A Formally Verified Optimizing Compiler. In: *ERTS 2016: Embedded Real Time Software and Systems*. 2016.

[Swa+16] Nikhil Swamy, Markulf Kohlweiss, Jean-Karim Zinzindohoue, Santiago Zanella-Béguelin, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, and Pierre-Yves Strub. Dependent Types and Multi-Monadic Effects in F*. In: 2016, pp. 256–270. DOI: 10.1145/2837614.2837655.

[Yan+16] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, Dynamic Information Flow for Database-Backed Applications. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016 PLDI '16, pp. 631–647. DOI: 10.1145/2908080.2908098.

[AR17] Maximilian Algehed and Alejandro Russo. Encoding DCC in Haskell. In: *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security (PLAS'17)*. 2017, pp. 77–89. DOI: 10.1145/3139337.3139338.

[ASF17] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. Multiple Facets for Dynamic Information Flow with Exceptions. New York, NY, USA. 2017. DOI: 10.1145/3024086.

[Bha+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Everest: Towards a Verified, Drop-in Replacement of HTTPS. 2017. DOI: 10.4230/LIPICS.SNAPL.2017.1.

[Bon+17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, Rustan Leino, Jay Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-Performance Cryptographic Assembly Code. In: *Proceedings of the USENIX Security Symposium*. 2017. DOI: 10.5555/3241189.3241261.

[Bra17] Edwin Brady. Type-Driven Development with Idris. 2017. ISBN: 9781617293023.

[Dar+17] David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting Definitional Interpreters. 2017. DOI: 10.1145/3110256.

[Jou17] Jacques-Henri Jourdan. Sparsity Preserving Algorithms for Octagons. 2017. DOI: 10.1016/j.entcs.2017.02.004.

[Pro+17] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified Low-Level Programming Embedded in F*. 2017. DOI: 10.1145/3110261.

[Ste+17] Deian Stefan, David Mazières, John C Mitchell, and Alejandro Russo. Flexible Dynamic Information Flow Control in the Presence of Exceptions. 2017. DOI: 10.1017/S0956796816000241.

[Zin+17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL: A Verified Modern Cryptographic Library. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017 CCS '17, pp. 1789–1806. DOI: 10.1145/3133956.3134043.

[Ahm+18] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a Witness: Foundations and Applications of Monotonic State. 2018. DOI: 10.1145/3158153.

[Fer+18] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1583–1600. DOI: 10.1145/3243734.3243743.

[Pal+18] Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P. Carloni. Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications. In: *2018 IEEE High Performance Extreme Computing Conference (HPEC)*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547578.

[Vas+18] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. Mac a Verified Static Information-Flow Control Library. 2018. DOI: 10.1016/j.jlamp.2017.12.003.

[Vaz+18] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement Reflection: Complete Verification with SMT. 2018. DOI: 10.1145/3158141.

[AB19] Maximilian Algehed and Jean-Philippe Bernardy. Simple Noninterference from Parametricity. 2019. DOI: 10.1145/3341693.

[Bes+19] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. Compiling Sandboxes: Formally Verified Software Fault Isolation. In: *Programming Languages and Systems*. 2019, pp. 499–524. DOI: 10.1007/978-3-030-17184-1_18.

[Fro+19] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A Verified, Efficient Embedding of a Verifiable Assembly Language. In: *Principles of Programming Languages (POPL'19)*. 2019. DOI: 10.1145/3290376.

[GTA19] Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov. A Dependently Typed Library for Static Information-Flow Control in Idris. In: *International Conference on Principles of Security and Trust*. 2019, pp. 51–75. DOI: 10.1007/978-3-030-17138-4_3.

[Mai+19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra Monads for All. In: *24th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2019. DOI: 10.1145/3341708.

[Mar+19] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In: *28th European Symposium on Programming (ESOP)*. 2019, pp. 30–59. DOI: 10.1007/978-3-030-17184-1_2.

[PVH19] James Parker, Niki Vazou, and Michael Hicks. LWeb: Information Flow Security for Multi-Tier Web Applications. 2019. DOI: 10.1145/3290388.

[Pro+19] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00064.

[Ram+19] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In: *USENIX Security*. 2019.

[Gua+20] Marco Guarnieri, Boris Köpf, Jose Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: Principled Detection of Speculative Information Flows. In: *Security and Privacy Conference*. 2020.

[IMO20] Andrej Ivašković, Alan Mycroft, and Dominic Orchard. Data-Flow Analyses as Effects and Graded Monads. 2020. DOI: 10.4230/LIPIcs.FSCD.2020.15.

[Mar+20] Jean-Joseph Marty, Lucas Franceschino, Jean-Pierre Talpin, and Niki Vazou. LIO*: Low Level Information Flow Control in F. 2020.

[Pro+20] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In: *IEEE Symposium on Security and Privacy*. 2020.

[Swa+20] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs. In: *25th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2020. DOI: 10.1145/3409003.

[ABH21] Maximilian Algehed, Jean-Philippe Bernardy, and Cătălin Hriţcu. Dynamic IFC Theorems for Free! In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 2021, pp. 1–14.

[FPT21] Lucas Franceschino, David Pichardie, and Jean-Pierre Talpin. Verified Functional Programming of an Abstract Interpreter. In: *Proceedings of the 28th Symposium on Static Analysis*. 2021 SAS'13. DOI: 10.1145/3484271.3489382.

[PEVale21] Project-Everest/Vale. June 30, 2021. URL: https://github.com/project-everest/vale (visited on 07/19/2021).

[Ras+21] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Programming and Proving with Indexed Effects. In submission. 2021. URL: https://www.fstar-lang.org/papers/indexedeffects/.

[HaskST] Control.Monad.ST. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Control-Monad-ST.html (visited on 09/13/2021).

[FStdLib] F* standard library. URL: https://github.com/FStarLang/FStar/tree/master/ulib (visited on 10/10/2021).

[Fou] Ethereum Foundation. CRITICAL UPDATE Re: DAO Vulnerability. URL: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/ (visited on 09/12/2021).

[JMR] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: Verifying Functional Programs Using Abstract Interpreters. DOI: 10.1007/978-3-642-22110-1_38.

[Jou+] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. DOI: 10.1145/2676726.2676966.

[NixPM] Nix Package Manager Manual. URL: https://nixos.org/manual/nix/stable/ (visited on 10/10/2021).

[EVEREST] Provably Secure Communication Software. URL: https://project-everest.github.io/ (visited on 04/10/2021).

Titre : Programmation vérifiée à l'intersection des types dépendants et de l'analyse statique

Mot clés : Types dépendants, interprétation abstraite, programmation vérifiée

Résumé : La programmation *dirigée par les types* ou *orientée preuves* consiste à écrire et prouver des programmes simultanément. Elle émerge grâce aux langages équipés de types dépendants, et permet une formidable qualité logicielle, au prix de temps passé à écrire des preuves. Inversement, l'analyse statique vise à inférer des propriétés en analysant des programmes existants.
Cette thèse étudie la façon dont les systèmes avancés de typage et l'analyse statique peuvent coopérer. Quant à l'analyse statique, nous nous focalisons principalement sur une théorie correcte d'approximation de programmes : l'interprétation abstraite. Notre première contribution démontre l'efficacité de la programmation orientée preuves (avec le langage F*) pour écrire des interpréteurs abstraits formellement vérifiés. De tels interpréteurs existent, mais requièrent une expertise avec les assistants de preuves et en interprétation abstraite, les rendant particulièrement inaccessibles. Notre approche ne nécessite que très peu de preuves manuelles (un ordre de magnitude inférieur en comparaison avec les travaux similaires) : notre implémentation est particulièrement concise et accessible.
Nous avons ensuite étudié l'hybridation d'interpréteurs abstraits et de monades de précondition la plus faible (WP). Notre approche instrumente des interpréteurs abstraits en des transformeurs de monades de WP.
Enfin, nous avons travaillé sur les bénéfices des types dépendants et du système d'effets de F* pour le contrôle de flux d'information (IFC). Nous présentons une librairie permettant de vérifier des politiques d'IFC de manière flexible, entre statique et dynamique.

Title: Verified Programming at the Intersection of Dependent Types and Static Analysis

Keywords: Dependent types, abstract interpretation, verified programming

Abstract: Dependently-typed languages allow for a new paradigm: *proof-oriented* or *type-driven* programming, consisting in writing a program, its specifications and proofs simultaneously. This yields the greatest quality of software, at the cost of manual proof effort. Conversely, *static analysis methods* aim at inferring properties by analyzing existing programs –usually written without proofs in mind.
This Ph.D. thesis studies how advanced type systems and static analysis methods can work cooperatively. As for the latter, we focus primarily on a theory of sound approximation: *abstract interpretation*. Our first contribution demonstrates the effectiveness of proof-oriented programming (with the F* language) for writing verified sound abstract interpreters. Such interpreters exist but understanding them requires expertise in both proof-engineering and abstract interpretation. Our approach yields an order of magnitude less explicit proofs, leading to a very concise and accessible implementation.
We then study how abstract interpretation and weakest-precondition (WP) monads could be hybridized, aiming at better type inference for F*. Our approach consists in turning abstract interpreters into WP monad transformers.
We finally look at the benefits of F* dependent types and *effects* for *Information Control Flow* (IFC). We present the design and implementation of a library allowing any combination of static and dynamic IFC verification.