Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)

Sylvain Boulmé

Habilitation à Diriger des Recherches
de l'Université Grenoble Alpes

Spécialité: Informatique

Formally Verified Defensive Programming
(efficient Coq-verified computations from untrusted ML oracles)

by **Sylvain Boulmé**

Publicly defended in hybrid mode on September 27, 2021 in front of this jury:

Reviewers

Andrew W. Appel	Professor at Princeton University
Sandrine Blazy	Professeur à l'Université de Rennes 1
Greg Morrisett	Professor, Dean of Cornell Tech

Examiners

Hugo Herbelin	Directeur de Recherche à l'Inria
Xavier Leroy	Professeur au Collège de France

Chair (Président)

Jean-François Monin	Professeur à l'UGA

version: September 28, 2021

Foreword

> *We are fortunate that during the last two decades, many talented scientists have built the mathematical infrastructure we need – the theory and implementation of logical frameworks and automated theorem provers, type theory and type systems, compilation and memory management, and programming language design. The time is ripe to apply all of these advances as engineering tools in the construction of safe systems.*
>
> Andrew W. Appel in "*Foundational Proof-Carrying Code*" [App01].

> *From a proof point of view, the main lesson learned from the* CompCert *experiment is the following. When formally verifying a complex piece of software [...], it is not realistic to write the whole software using exclusively the specification language of the proof assistant. A more pragmatic approach [...] consists in reusing an existing implementation in order to separately verify its results.*
>
> Sandrine Blazy et al. in "*Formal Verification of a C Value Analysis [...]*" [Bla+13].

> *Abstract types are an invaluable tool to software designers. They aid programmers in reasoning about interfaces between different pieces of code. Despite this utility, it is very hard to prove that the informal reasoning about abstract types [...] is correct. One approach is to [...] appeal to results on parametricity [Rey83].*
>
> Dan Grossman, Greg Morrisett et al. in "*Syntactic Type Abstraction*" [GMZ00].

This document is my habilitation thesis ("Habilitation à Diriger des Recherches" or HDR, in French). It rewrites most of my published and unpublished papers since [#FB14], in order to make them appear as the parts of a single topic abstracted below.

How to Read this Document?

This document is best viewed in color, A4 paper, but a black and white print is also readable. The online PDF contains hyperlinks in red. Chapters, ended by a "†" (dagger) symbol in their title, are (totally or partially) extracted from my peer-reviewed publications that are indicated in a footnote in the first page of the corresponding chapter. These self citations start with character "#" like [#FB14]. They are not listed in Bibliography, but in Chapter 8. I also contributed to the formally verified software with names appearing inside braces, like {VPL}. These software are also summarized in Chapter 8.

Abstract

I present a *lightweight approach* – combining Coq and OCaml typecheckers – in order to *formally verify* higher-order imperative programs for *partial correctness*. In this approach, called FVDP (Formally Verified Defensive Programming), external OCaml functions are abstractly embedded in Coq as *nondeterministic untrusted oracles*: the formal proofs only consider their ML type, but never their side-effects nor other functional properties.

Formal guarantees are obtained by combining *parametric reasoning over polymorphic oracles* (i.e. "theorems for free" à la Wadler) with *verified defensive programming* in Coq. In particular, this combination is exploited within a design pattern—for certificate producing oracles—called *Polymorphic LCF Style* (or *Polymorphic Factory Style*). Large Coq proofs on these higher-order impure defensive computations are decomposed thanks to *data-refinement* techniques in order to cleanly separate reasoning on pure data-structures and algorithms from reasonings on sequences of impure computations. Then, the latter are (semi)automated thanks to computations of *weakest liberal preconditions*.

FVDP is detailed on several "realistic" applications: in optimizing compilation (instruction scheduling with {CompCert-KVX}), in static analysis (abstract domain of convex polyhedra with {VPL}) and in automated deduction (Boolean SAT-solving with {SatAnsCert} and linear rational arithmetic with {VplTactic}). The document explains how FVDP both alleviates development times and running times of such formally verified applications.

Acknowledgments

I am deeply honored that Andrew Appel, Sandrine Blazy and Greg Morrisett have accepted to review my manuscript. Their works are a great source of inspiration for my own. I am also very grateful to Xavier Leroy, Hugo Herbelin and Jean-François Monin for being in the jury. My work is indeed particularly indebted to Xavier, who created OCaml and CompCert, and who has moreover found time for providing me useful answers, feedbacks and advice. Obviously, it is also largely indebted to Coq developers, represented in the jury by Hugo. The first edition (in French) of Jean-François's book on formal methods [Mon03] helped me to get a broad view of the field when beginning my Phd thesis: twenty-five years later, my habilitation thesis, by combining several paradigms from distinct formal methods, is still under the influence of his book.

Substantial parts of this document have been co-written with my main coauthors: Alexandre Maréchal, David Monniaux and Cyril Six. Many important ideas of this work – "verified programming with oracles" as a research project, applying theorems for free, and refinement for abstract interpretations, etc – have emerged in informal chats with Michaël Périn. Alexis Fouilhé and Thomas Vandendorpe have largely contributed to the development of the case studies presented here: Alexis developed the first version of the {VPL} (in his Phd [Fou15]), and Thomas wrote the OCaml oracles of {SatAnsCert} in his bachelor internship. Marie-Laure Potet encouraged me to write this document and introduced me to computer security. Several years ago, she also helped me to better understand data refinement. I am indepted to many other people: some complementary acknowledgments are in the French part.

Description en français

Traduction du titre

Programmation défensive formellement vérifiée :
calculs efficaces et vérifiés en Coq,
à partir d'oracles OCaml potentiellement non fiables

Résumé détaillé

Ce mémoire de HDR présente une *approche légère* – combinant Coq et OCaml – afin de vérifier formellement des programmes impératifs d'ordre supérieur en *correction partielle*. Dans cette approche, que j'appelle FVDP pour "Formally Verified Defensive Programming" (la programmation défensive formellement vérifiée), le logiciel, formellement prouvé en Coq, comporte aussi des fonctions directement programmées en OCaml, considérées comme des *oracles potentiellement non fiables*, c'est-à-dire dont l'implémentation est ignorée par la preuve formelle, mais dont les résultats sont éventuellement vérifiés à l'exécution, de façon à garantir les propriétés formelles désirées.

Ainsi, pour obtenir un programme formellement vérifié résolvant un problème compliqué, le plus simple consiste souvent à déléguer à un oracle OCaml la recherche d'une "bonne" solution, en le combinant en Coq avec un test défensif (formellement vérifié) capable de garantir que cette solution satisfait la propriété de correction désirée. L'intérêt est double : d'une part, bénéficier d'un langage de programmation plus riche que celui de Coq pour la recherche de solution (avec toute la puissance de la programmation impérative en OCaml); d'autre part, éviter d'avoir à formaliser les raisonnements sur les procédés de calculs de la solution, souvent beaucoup plus complexes que ceux juste nécessaires à la vérification défensive de cette solution. Ainsi, la preuve formelle en Coq ne considère les oracles qu'au travers de leur type OCaml (via son plongement en Coq). En particulier, elle ne raisonne jamais sur leurs effets de bord.

Le mémoire retrace brièvement l'histoire de cette idée, apparue bien antérieurement à mes travaux et dans un contexte plus large que le couplage Coq/OCaml. Il rappelle en particulier comment elle a contribué au succès de CompCert [Ler09a], un compilateur C formellement vérifié en Coq.

Puis, il propose de systématiser le couplage Coq/OCaml précédemment utilisé par CompCert, basé sur l'extraction de Coq vers OCaml. Il part du constat, qu'en général, pour éviter un risque d'incorrection, il faut plonger les oracles OCaml en Coq comme des fonctions "non-déterministes". J'introduis pour cela une monade dédiée, appelée "may-return monad". Cette structure permet de combiner ces fonctions non-déterministes et de raisonner sur leurs résultats potentiels. Elle est à la base du prototype d'interface Coq/OCaml fournie par ma bibliothèque appelée {Impure}. Cette dernière propose en outre un plongement de l'égalité de pointeurs d'OCaml en Coq via une telle

fonction non-déterministe. Le mémoire explique comment cette simple égalité de pointeurs permet la vérification défensive légère d'oracles pratiquant de la mémoïsation dans des tables de hachage ("hash-consing" de termes, récursion avec mémoïsation).

Le mémoire détaille aussi comment le type OCaml suffit à garantir statiquement des invariants expressifs sur le résultat des oracles *polymorphes*. Techniquement, cela correspond à relâcher la technique des "*theorems for free*" de Wadler [Wad89] (basée sur le *polymorphisme paramétrique* de Reynolds [Rey83]) dans un contexte impératif et non-déterministe. Cette technique est notamment à la base d'un patron de conception destiné aux oracles devant fournir une trace vérifiable de leurs calculs (cette trace devant par exemple permettre au vérificateur de démontrer la non-existence d'une solution à un problème donné). Ce patron de conception, que j'appelle "*Polymorphic Factory*" (la fabrique polymorphe) ou encore "*Polymorphic LCF Style*" (en référence à l'ancien prouveur LCF [Mil79] à l'origine des langages de programmation ML), combine donc le typage statique de OCaml et de Coq pour alléger à la fois le développement de ces oracles et leur vérification défensive.

Enfin, je propose de décomposer certains imposantes preuves Coq sur ces programmes formellement non-déterministes, avec des techniques de *raffinement de données*. Cela permet d'isoler proprement les raisonnements sur les structures de données et les algorithmes "purs", des raisonnements sur les séquences d'appel de fonctions non-déterministes. Ces derniers peuvent être alors (semi)automatisé grâce à un *calcul de plus faible précondition* (fourni par {Impure}).

Le mémoire illustre ces idées sur plusieurs applications "réalistes" : en compilation optimisante (ordonnancement d'instructions avec {CompCert-KVX}), en analyse statique (domaine abstrait de polyèdres convexes avec la {VPL}) et en déduction automatisée (en SAT booléen avec {SatAnsCert} et en arithmétique rationnelle linéaire avec {VplTactic}). Il détaille comment la FVDP, déclinée dans des conceptions logicielles adaptées, allège à la fois les temps de développement et d'exécution de ces applications formellement vérifiées.

Remerciements complémentaires (à ceux du "Foreword")

Cette thèse d'habilitation n'aurait pas pu voir le jour, sans la délégation au CNRS dont j'ai bénéficié sur l'année universitaire 2020-2021. Je remercie Florence Maraninchi, alors directrice du laboratoire Verimag, et Jean-Louis Roch, directeur de l'Ensimag (où j'enseigne), d'avoir appuyé ma candidature à cette délégation. Je suis aussi par conséquent lourdement redevable aux collègues qui ont dû me remplacer au pied levé, malgré leurs charges déjà lourdes, dans une situation épidémique terrible, notamment Xavier Nicollin, Matthias Ramparison, Patrick Reignier et Lionel Rieg. Je suis enfin redevable à tous les collègues de Verimag, administratifs, techniciens, ingénieurs, enseignants, chercheurs, qui animent et font tourner le labo, et qui *in fine* contribuent à mon travail de recherche au quotidien depuis des années.

Je remercie aussi les étudiants qui ont travaillé cette année avec David, Cyril et moi sur notre version de CompCert : Justus Fasse, Léo Gourdin, Lucas Chaloyard, Pierre Goutagny et Nicolas Nardino. Votre enthousiasme pour ce projet (malgré les restrictions sanitaires) a été un moteur de ma motivation. Vous nous avez aussi permis de bien progresser (même si cela n'apparaît pas dans ce document).

Je remercie enfin ma famille et mes proches d'avoir permis que ma charge mentale soit accaparée pendant des mois par des questions bien loin de leur quotidien.

Contents

Chapter 1

Introduction to Formally Verified Defensive Programming (FVDP)

By June 1949 people had begun to realize that it was not so easy to get programs right as at one time appeared. I well remember when [...] the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

Maurice Wilkes, Turing Award[1] 1967, in "*Memoirs of a Computer Pioneer*", 1985.

We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as Very Humble Programmers.

Edsger W. Dijkstra, in conclusion of his Turing Award[2] lecture [Dij72].

Contents

[1] For having designed the first computer with an internally stored program : the EDSAC, in 1949.
[2] For its seminal contributions to the science and art of programming.

This preliminary chapter intends to introduce—for a wide audience of computer scientists (without any prior knowledge about formal verification in Coq)—both the scientific context and the contributions of my work.

1.1 How to Protect Programs Against Programming Errors?

The issue faced by 1949 programmers is still open today: it is still hard *to get programs right.* Of course, software engineers are much more equipped today than a few decade ago (e.g. with myriads of testing methodologies [Mat08]). But software and computers are also much more complex. And mainstream methods, while very effective in many contexts, cannot ensure the quasi-absence of bugs that is required for safety-critical embedded systems (e.g. automated transport, power plants, etc). Quoting Dijkstra [Dij72]: *program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.*

This document studies a method—that I call *Formally Verified Defensive Programming* or FVDP in short—for efficiently programming "*software-handling software*" (e.g. compilers or verifiers), while *formally proving* their *functional correctness*. FVDP consists in *monitoring* at runtime the results of some untrusted but efficient computations (this is defensive programming), and in formally proving that this runtime monitoring ensures the expected correctness property. Hence, FVDP strengthens defensive programming in order to formally ensure the correctness of computation results. More precisely, FVDP aims to ensure *the partial correctness* of programs: if the programs succeeds to return a result, then this result is correct. For example, if the compiler succeeds in producing an executable, then the executable behaves as expected (and can be safely embedded in the safety-critical system). Quoting Leroy [Ler11]:

> *Not all parts of a compiler or verifier need to be proved: only those parts that affect soundness, but not those parts that only affect termination, precision of the analysis, or efficiency of the generated code. Leveraging this effect, complex algorithms can often be decomposed into an untrusted implementation followed by a formally verified validator that checks the computed results for soundness and fails otherwise. (Failure is not an option in flight, but is an option at compile-time and verification-time.)*

Indeed, FVDP has been successfully applied in the CompCert verified compiler [Ler09b; Ler09a]. This compiler is the first C compiler used in industry [Bed+12; Käs+18] that provides a formal proof of correctness. It is a major success of software verification, because CompCert does not have the bugs which can usually be found in optimizing compilers [Yan+11]. Its success partly comes from its smart design, focusing the formal proof in Coq [Tea20] on the partial correctness of compilation passes, while reasonings on their performance and termination remain informal.[3] In particular, CompCert

[3] Other key ingredients of CompCert success story include for example its nice memory model [LB08; Ler+14] allowing for stepwise refinement of the memory layout through the compiler passes.

invokes *oracles*, i.e. untrusted functions programmed in OCaml [Ler+20], from the certified code. Formal guarantees are proved from *runtime checks* on *untrusted intermediate results*. For example, register allocation in compilers is an NP-complete problem: efficiently finding a fitting allocation is difficult, while checking the validity of a given one is easy. CompCert thus delegates complex parts to an untrusted oracle, the result of which is then validated by a checker programmed and certified correct in Coq [RL10].[4]

As advocated by Leroy, such a decomposition—into an untrusted oracle and a formally verified checker—may greatly alleviate formal verification of many complex implementations, by sparing painful proofs, about their termination, about their memoization techniques of redundant computations, about their strategies to select solution candidates, etc. Generally, untrusted oracles have several benefits: (1) they avoid the implementation and proof of cumbersome algorithms in Coq; (2) they offer the opportunity to use (or even reuse) efficient imperative implementations; (3) above all, they make the software more modular since the checker is actually certified for any oracle with the same type. In particular, the oracle can still be improved or tuned for some specific cases, without requiring any update in the checker.

FVDP does not try to prevent all bugs. On the contrary, FVDP can be viewed as a systematic approach to convert "*correctness bugs*" into "*abnormal termination*", such that correctness bugs cannot remain unnoticed. This feature could be viewed as a weakness. But this weakness seems a lesser evil for software that produces safety-critical software: while the safety-critical software is not produced, it cannot cause harmful damage. The benefit of this modest protection is a great simplification of the formal verification task, which in turn helps to tackle more complex applications, as demonstrated by CompCert register allocation.

Because FVDP does not prevent "*performance bugs*" (which include here abnormal termination and non-termination), FVDP still require the use of standard testing techniques in order to check that the software is reasonably usable. Moreover, the main benefit of FVDP is precisely to mitigate the previously mentioned claim of Dijkstra about program testing: *program testing, in conjunction with FVDP, is a very effective way to show the absence of bugs in the results returned on the tested inputs, particularly if there is no other easy way to check that these results are correct*. However, studying in details how testing may benefit from FVDP (and conversely) is out of the scope of this document: in a first approach, we may simply use FVDP as a replacement of unverified defensive programming.

This document presents applications of FVDP by combining the Coq proof assistant [Tea20] and the OCaml programming language [Ler+20]. Actually, FVDP has already been applied in many pre-existing works, under distinct names: "*skeptical approach*" [HT98], "*translation validation*" [Nec00], "*result certification*" [Bes+07] or "*validation a posteriori*" [Ler11]. I introduce instead the name "*formally verified defensive programming*", in order to suggest that a systematic application of this approach could increase the productivity of formally verified programming. FVDP may indeed avoid many tedious proofs, by replacing them with simple defensive checks, provided that these defensive checks are cheap enough at runtime. The following paragraphs aim to clarify and motivate FVDP's intrinsic limitations.

[4] Remark that a purely functional implementation of the algorithm behind the untrusted oracle has also been verified in Coq [BRA10]. But it is not as fast as an imperative one. And, it might also not be as easily tunable.

```
Fixpoint iterp2 {A} (f:A → A) (n:nat) (x:A): A :=
  match n with
  | O ⇒ f x
  | S n ⇒ iterp2 f n (iterp2 f n x)
  end.
Definition power2 n := iterp2 S n O.
Definition bigtower2 n := iterp2 power2 n O.
Definition loop n := iterp2 (fun x ⇒ x) n n.
```

Figure 1.1: Example of the Counterintuitive Meaning of "Total Correctness".

1.1.1 Without Protection Against Non-Termination and Crashes

While *autarkic* programming[5] in Coq ensures *total correctness*, FVDP only ensures *partial correctness*, due to its use of external untrusted code which may fail or loop forever. Actually, we now illustrate that, for the final user, the usual notion of "total correctness" in computer science does not provide much more tangible guarantees than "partial correctness".

Indeed, while Coq ensures that all typechecked programs terminate (without error), this happens for the usual mathematical notion of termination in computer science which does not match the usual expectation of final users: a Coq program, that is proved to terminate (without error), may, in practice, not finish before the end of universe, or may crash because of a lack of memory. Indeed, the mathematical notion of termination assumes unbounded time and memory.[6] Let us illustrate this on the simple examples of Fig. 1.1. First, let me explain its definitions, based on the inductive definition of a Peano's natural number `n` from zero `O` and successor `S`. The structurally recursive `(iterp2 f n x)` computes $(\texttt{f}^{2^n}\ \texttt{x})$. Hence, mathematically,

- `(power2 n)` computes 2^n;
- `(bigtower2 n)` computes a 2^n-height tower of powers of the form $2^{\overset{2^{\cdots^{2^0}}}{\}2^n \text{ times}}}$;
- at last, `loop` is a very inefficient implementation of the identity function over naturals: `(loop n)` returns `n`.

The Coq logic ensures that all these computations terminate. But, on my laptop:[7]

- `(power2 (bigtower2 (S (S O))))` which is expected to compute $2^{16} = 65536$, actually crashes almost instantaneously on a stack overflow;
- `(bigtower2 (S (S (S O))))` which is expected to compute $2^{2^{2^{65536}}}$ actually crashes after almost 3 minutes on a memory overflow;
- `(loop (power2 (S (S (S (S (S (S (S (S (S (S O)))))))))))))` cannot be observationally distinguished from an infinite loop: indeed, it applies 2^{1024} times the identity, while the age of observable universe is estimated to be lower than 2^{90} nanoseconds.

These bad behaviors are not due to a bug in the implementation of Coq, but result from the mismatch between the mathematical notion of termination and the human one. From the final user point's of

[5] Following [BB02], we say that a program developed in Coq is *autarkic*, iff it is only programmed within the Coq programming language, without any external code. We also say—as a synonym—that such a program is *pure*, because it can only correspond to a mathematical function, without any implicit side-effect (such as an I/O event or the modification of a variable).

[6] The motivation for this abstract notion of termination lies in the design of Coq logic: in short, ensuring such a termination of computations is necessary to unify the representation of proofs and computations by the Curry-Howard isomorphism. Actually, this design of Coq improves the trustworthiness of Coq developments. See Section 1.2.1.

[7] The computations here have been run within the `coqc` compiler. Similar bad behavior could be also observed after extraction/compilation to native code.

view, the Coq proof assistant can only ensure a partial correctness property. It is technically hard to ensure stronger properties. For example, safety-critical systems require strong termination properties on computations: they are expected to fit in the finite memory of the processor and to finish before any significant change of the environment. In particular, some static analyzers (like aiT[8], see [Sch+18]) are able to bound the worst-case execution time (WCET) of embedded binaries, by using a fine model of the processor micro-architecture [Fer+01; Mai+19].[9] But, such analyzes are very specific to the target processor and to embedded software. They seem almost impossible to generalize to general purpose computations in Coq. As a consequence, dropping a quite confusing "total correctness" in favor of an uninhibited "partial correctness" does not seem a big loss.

1.1.2 Without Protection Against Malicious Bugs

The FVDP method studied here will provide no absolute guarantee: even against the "correctness bugs" that it aims to prevent. The method only aims to make unintended correctness bugs very unlikely. Indeed, no method can provide absolute guarantees. This is related to Gödel's second incompleteness theorem [Göd31]: even if we aim to carefully check our mathematical theories, this check will remain incomplete—because it cannot completely bootstrap itself—and may obfuscate an error. Fundamentally, we can only have an incomplete knowledge of our surrounding reality which forbids to us to fully check the correctness of our mathematical models: *our ignorance may maintain us in wrong beliefs.*

The intuition of this issue becomes clearer when we imagine that the incompleteness of our knowledge is maliciously exploited at our expense, like in popular movies such as "*The Matrix*" or "*The Truman Show*". In 1983, Ken Thompson and Dennis Ritchie jointly received the Turing Award for their implementation of the UNIX operating system. Thompson's acceptance speech [Tho84] was precisely on such a subject: how to introduce Trojan horses (e.g. secret thieves) in UNIX systems, such that these backdoors remain invisible in the source of programs, but are only present in their binaries. Indeed, initially, users are somehow obliged to install the OS on their computer from some binaries. Thompson's trick is that, even if users later recompile programs from the sources, this will be achieved with the binary of a C compiler, which contains itself a *malicious bug* producing Trojan horses. Recompiling the C compiler itself without changing the sources does not make disappear the backdoor. This backdoor being hidden in a given sequence of characters of the compiler sources which seems harmless for the C semantics, there is no reason for the user to change it. Hence, Thompson's malicious C compiler does deliberately fail to respect the C semantics: it maliciously interprets this special sequence of characters. And Thompson concludes:

> *No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the* C *compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect.*

Indeed, since Thompson's seminal talk, malicious hardware has been designed (e.g. see [Kin+08]). Protections against such a malicious hardware have been proposed by [Zha+13] and [WSS13]. But then, counterattacks against these protections have been designed [ZYX14], etc.

[8] https://www.absint.com/ait/

[9] See also [Oli+14] for the FVDP of a WCET analysis within CompCert.

Hence, Thompson's malicious bugs really exist. Actually, in the usual metatheories of program verification, nothing distinguish a malicious bug from an unintended bug: *a malicious bug simply corresponds to a bug that seems so unlikely, that it should not happen by accident*. Moreover, to Thompson's list of "program-handling programs" given above, we may also add programs to build correct programs, like the Coq proof assistant (which, in a sense, simply extend the static-typechecking process of the compiler). Thompson has thus sketched the proof of another incompleteness theorem: *a program verifier cannot provide absolute guarantees on its own implementation.*

For instance, Barras [Bar99] and Sozeau et al. [Soz+20] have shown how to mitigate Gödel's incompleteness theorem: by only admitting as an axiom the strong normalization of Coq rewriting rules (which entails the consistency of its logic), they succeed to prove the correctness of an implementation of the Coq kernel with Coq. But, as sketched by Thompson, their proofs only prevent from unintended bugs by Coq developers. Some developer may still imagine an incorrect implementation that is enough malicious to prove itself with a correct proof – such that the incorrectness can only be detected by scrutinizing its binary representation. This negative remark does not discredit the great value of such correctness proofs. Because, when applied with a correct prover, the vicious circle disappears: exactly like when compiling Thompson's C compiler with a correct C compiler. Thompson's bug is not in the source, it is in the binary.

1.1.3 With Formal Protection Against Unintended Correctness Bugs

FVDP protects against unintended bugs in untrusted computations. For example, in some cases, an untrusted computation could silently corrupt the memory of verified monitors in order to make them accept wrong results. But, it seems very unlikely that such a memory corruption results from an unintended bug. It is much more probable that an unintended memory corruption leads the software to crash, which is not considered in FVDP as a correctness bug. In other words, even if FVDP assumes untrusted oracles to be memory safe[10], in practice, FVDP still provide strong protections even in case of unintended memory corruptions. Such unintended memory corruption may typically occur when embedding C code within OCaml oracles, through the OCaml/C Foreign Function Interface (FFI).

1.2 Analyzing the Trusted Computing Base (TCB) for FVDP

In short, Thompson [Tho84] simply illustrates that the safety/security of every computing system relies on some of its subsystem which needs to be trusted. An important concern is thus to identify this *Trusted Computing Base* (TCB), defined by Lampson et al. [Lam+91] as:

> *[the] small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.*

When applied to automated verification, we simply replace "security" by "correctness" in the above definition of TCB. Having a "small" TCB is one criterion among others to appreciate a verification method (e.g. see [App+03]). Another one is its ability to improve the safety of large, complex and useful software. This section gradually introduces what possible TCBs are required for FVDP based on Coq and OCaml, depending on the way to connect the Coq and OCaml codes.

[10] In particular, untrusted oracles, that are implemented by type safe OCaml code, are memory safe: this is guaranteed by OCaml type-checker.

1.2.1 TCB of Coq Proof Checking

Let us consider the TCB of the Coq proof assistant. The Coq engine is based on a small *kernel*: the typechecker of *proof terms* in an extension of the Calculus of Inductive Constructions [CP88; Tea20]. Within an interactive proof, the user may invoke untrusted tactics, but these tactics must produce proof terms; at the end of the proof, a whole proof term is produced and finally checked by this independent kernel. This defensive design of the prover, separating a small well-delimited trusted kernel from convoluted untrusted tactics, is known as the *de Bruijn criterion*, in reference to the design of Automath, one of the first proof assistants [dBru68].

Within the `coqc` compiler (or `coqtop`/`coqide` interpreters), this defensive design offers a good protection against unintended bugs in tactics. But it does not prevent "malicious" bugs in *plugins*, which are tactics implemented in OCaml (the implementation language of Coq tools themselves), often programmed by third-party users, and dynamically loaded at runtime. By this way, such a third-party malicious programmer may introduce some unsafe OCaml code, able to silently corrupt the kernel state, in order to make it accept wrong theorems.

However, at the end, `coqc` compiles the user proof terms into a binary format ("`.vo`" libraries). These compiled proof terms can themselves be independently checked with a small checker, called `coqchk`, almost reduced to the Coq kernel, without any invocation of potentially malicious tactics.[11] Finally, the TCB of the Coq proof assistant thus reduces to the one of `coqchk` executable (within the underlying operating system and hardware). As illustrated by [Bar99; Soz+20], this program is sufficiently small to be itself formally verified.

In practice, the TCB of a formally verified development is not reduced to the TCB of its formal verification: it also includes a part of the development itself, called here the "*specification*", that consists in the formal description of the (external) objects on which this development claims to prove properties. Section 1.2.5 details this part of CompCert TCB. Here, let us simply explain how the Coq design helps to design trustworthy specifications. First, the expressiveness of the logic helps users to design Coq theories by using *definitions* instead of *axioms*. Indeed, a user theory defined without axioms is sure to be consistent (i.e. without contradiction): more precisely, its consistency is a consequence of the strong normalization of the Coq rewriting system (i.e. termination of computations). In practice, users may also safely invoke standard axioms (like functional extensionality) which are known to be consistent with the Coq logic. Second, users may also design *executable* specifications. This helps to validate their accuracy with respect to what they are modeling. For example, [KW15] defines several formal models of the C programming language in Coq and prove their equivalence. One of them is executable and is validated on actual C programs, while others help to reason on C programs in Coq.

For a more detailed presentation, see [AI20]: it provides guidelines on the use of Coq for computer security certification by Common Criteria.

1.2.2 TCB of Coq Verified Autarkic Programs

Let us now consider the minimal TCB required by verified autarkic[5] programming within the Coq proof assistant. Actually, the TCB depends on whether the verified programs are run inside a given Coq proof, or whether they are natively run, after having being extracted toward OCaml (or Haskell) and then compiled into native code. In the first case, the TCB is similar to the TCB analyzed in Section 1.2.1 (i.e. the TCB of `coqchk` and the specification of the considered development). In the

[11] Moreover, the user can load these binary files with `coqtop` or `coqide`, in order to "manually" check that the statements, which are here proved, correspond to those expected.

second case, the minimal TCB is a bit larger: it also includes the Coq extraction process itself, and also the OCaml (or Haskell) compiler.[12] This second execution mode is however much more efficient to run large software, in particular, because all the subterms that are only relevant for proofs but not for computations have been pruned. Actually, Coq extraction is a quite complex compiler [Let04; Soz+20].

1.2.3 TCB of FVDP with Coq and OCaml through Extraction

Another interest of extraction is that it can be easily instrumented, in order to combine impure computations with pure ones as demonstrated by CompCert. Hence, CompCert uses a standard FFI (*Foreign Function Interface*) of the Coq programming language, in order to invoke external OCaml code from Coq code. However, currently, there is no formal justification regarding the soundness of this FFI.

Chapter 2, extending the work started in [#FB14], investigates the unsoundness that can arise from such a use of OCaml oracles through Coq extraction and propose solutions to avoid them. It aims to keep the TCB of FVDP almost as small as the TCB of autarkic Coq programs through Coq extraction, except that the OCaml typechecking of oracles is critical for correctness. In contrast, Coq extraction is authorized to completely bypass OCaml typechecking (e.g. through "`Obj.magic`"), because the correctness of extracted code is ensured by Coq typechecking. Thus, it seems relevant to state that FVDP software like CompCert are verified with the Coq+OCaml proof assistant. However, fully formalizing the foundations of this cooperation between Coq and OCaml typechecking is left for future works.[13]

1.2.4 Toward Removing Extraction and OCaml from the TCB

Letan and Régis-Gianas [LR20] propose to combine impure OCaml computations and Coq computations by using the plugin mechanism instead of extraction, in their FreeSpec[14] plugin. Their approach provides mechanisms: (1) to specify in Coq *formal contracts* on external imperative components; (2) to combine in Coq these components and prove properties *from the contracts* about these combinations; (3) to run these impure computations inside the Coq engine. For example, Letan and Régis-Gianas [LR20] program, verify and run a mini HTTP server within Coq.

Their approach is the opposite of FVDP: they use external imperative components as *trusted* components instead of *untrusted* ones. But, some of their techniques could be probably reused in order to build Coq proofs reflecting a given run of programs like CompCert that interleaves external untrusted impure computations and verified pure ones. In other words, we could probably alleviate the TCB of FVDP by (1) basing the FFI of Coq on a plugin instead of extraction; (2) encapsulating runs of these programs within proofs; (3) removing the need of trusting this Coq plugin by checking these proofs with `coqchk`.[15] Actually, the first goal of this document is to study the power of FVDP in formal proofs of "realistic applications". It is thus focused on making FVDP as lightweight as possible. Replacing extraction by a plugin mechanism makes the framework both much more complicated and much more heavyweight at runtime. Thus, this document only marginally explores this possibility (see Chapter 6).

[12] We may consider that the TCB of Coq already includes OCaml, since the Coq checker is written in OCaml. But, other implementations of the Coq checker could also exist.

[13] The subject seems too ambitious for me on my own: if you are interested in, please contact me.

[14] https://github.com/ANSSI-FR/FreeSpec

[15] Such a system would look like an extension of Cybele (https://github.com/clarus/cybele) able to deal with arbitrary external OCaml oracles. See [Cla+13] and Section 2.6.

1.2.5 TCB of Large and Complex Software like CompCert

The TCB of a software like CompCert which is expected to interact with a complex computing environment is *much larger* than the one of its verification method. And actually, the most *critical parts* of CompCert TCB do not derive from its verification method, but from its formal model of this complex environment and a few trusted (and not formally verified) components. In particular, an unintended bug in Coq extraction or in the OCaml compiler is probably not critical for CompCert; it is likely to only break down its compilation or at worst to lead it to crash at runtime (whereas crashes in software produced by CompCert are critical, crashes in CompCert itself are not). Here are the parts of CompCert TCB which do not directly derive from its verification method in Coq/OCaml:

- the formal semantics of the first formal language, called CompCert C (in Coq);
- the formal semantics of a formal assembly language (in Coq);
- option parsing and filename handling (in OCaml);
- the preprocessor (partly external, partly in OCaml), which turns regular C into CompCert C and optionally deals with some constructs (bitfields, passing and returning structures to functions, variable length arguments...);
- the "assembly expansion" pass (in OCaml) that expands certain pseudo-instructions into actual assembly code, including: stack allocation, stack deallocation, memory copy;
- the formal axiomatization (in Coq) of these pseudo-instructions (e.g., the registers they may clobber);
- the assembly language printer (in OCaml);
- the compatibility of the application binary interface used by CompCert with that of the compiler used to compile other libraries on the system, including the standard C library;
- the external assembler and linker.

Since the beginning of CompCert, the trustworthiness of the frontend has strongly increased, with a formal validation of the parser [JPL12], and with a very careful comparison of the formal CompCert C semantics to the C standard (partly mechanized with the help of the CompCert C reference interpreter) [KLW14; KW15]. However, reducing the above TCB, especially on the backend, still remains a challenge for CompCert developers. See for example this recent line of works, refining the semantic models of CompCert toward more realistic ones, but not yet integrated into the "official" releases: [BBW19b; BBW19a; WWS19; Wan+20].

Since CompCert's origin almost 15 years ago, some critical bugs have been discovered in it (mostly discovered by CompCert developers themselves, with standard testing techniques). To my knowledge, all of them were either located in its Coq formal specifications or in its trusted (but not formally verified) OCaml components. None of them is related to the few critical bugs discovered in the meantime within the Coq and OCaml implementations. None of them is related to the foundation-lacking usage of untrusted OCaml oracles (which is detailed by Section 2.1). In other words, the formal verification of CompCert with Coq and OCaml is a great success, since no bug has been found in the verified components of CompCert, even though these verification tools may not be themselves perfectly correct. CompCert demonstrates that FVDP is relevant for producing safer useful software.

1.3 Design Principles for FVDP

We now present some important concepts at the basis of our FVDP designs. Section 1.3.1 introduces various strategies for embedding (or not) oracles within Coq. Section 1.3.2 introduces a well-established technique to decompose large formally verified developments.

1.3.1 Communicating (or not) with Untrusted Oracles from Coq-Verified Code

The idea to reduce the TCB of rich logical frameworks to a small proof checker—called the kernel—was pioneered in the design of two interactive provers, Automath [dBru68] and LCF (which stands for *Logic for Computable Functions*) [GMW79]. But, their style is very different. Automath introduces a notion of "*proof object*" and implements the kernel itself as a typechecker, thanks to the Curry-Howard isomorphism. In contrast, LCF is written as a library in a functional programming language—called ML for "Meta-Language"—which provides the type of theorems as an abstract datatype [Gor+78]. Its safety relies on the fact that objects of this type can only be defined from a few primitives (i.e. the kernel). Each of them corresponds to an inference rule of Higher-Order Logic in natural deduction. LCF style is much more lightweight – both for the development and the execution of proof tactics – whereas the proof object style allows for a richer logic, embedding quite arbitrary computations within the type system (e.g. using *dependent types*). Nowadays, the kernel of skeptical interactive provers is still designed according to one of these styles: Coq has proof objects whereas HOL provers are in LCF style. Moreover, some descendants of ML—like OCaml—have integrated powerful impure programming features without damaging ML type safety.

With Maréchal [Mar17], we have proposed to classify Coq-verified computations in three styles of design. Figure 1.2(*a*) illustrates the simplest approach for "simple" computations: the computation is directly implemented and proved correct in Coq following the *autarkic* style advocated by Barendregt and Barendsen [BB02]. However, quoting Harrison and Théry [HT98], when the computation becomes "sufficiently" complex (e.g. when using dynamic programming techniques is desirable, or when solving NP/co-NP hard problems, etc)

> *the separation of proof search and proof checking offers an easy way of incorporating sophisticated algorithms, computationally intensive search techniques and elaborate heuristics, without compromising either the efficiency of search or the security of proofs eventually found.*

This is the motivation for introducing a *defensive design*. Roughly speaking, such a design simply replicates the one of proof assistants within applications.

Hence, we distinguish two defensive styles for Coq. The first one consists in defining an intermediate *format of witness*, while delegating to an external OCaml oracle the *search* for a witness that will drive a Coq-verified computation to the expected result. When this witness format becomes itself quite complex: we rather call it a *certificate language*, as illustrated in Figure 1.2(*b*). This style is mandatory in order to invoke FVDP computations for simplifying Coq interactive proofs (i.e. within Coq tactic by computational reflection). Hence, Coq plugins provide many instances of this style: see the famous Micromega tactics [Bes06; Tea20], SMTCoq [Kel13], or Cybele [Cla+13]. An example is also here provided in Chap. 6.

The second one—illustrated in Figure 1.2(*c*)—relies on Coq extraction to apply LCF style within the OCaml oracle. Instead of introducing an abstract syntax in order to represent certificates which themselves drive certified computations, this style makes the OCaml oracle directly build correct-by-construction results through a Coq-certified factory. In other words, Figure 1.2(*b*) corresponds to use a *deep embedding* of the certificate language (both in Coq and OCaml), whereas Figure 1.2(*c*) corresponds to use a *shallow embedding* of this language. This document will explain why shallow embeddings are often more lightweight than deep ones (both in development and running times).[16] But in our context, LCF style requires Coq extraction and relies on the correctness of OCaml type system, which may not be an option, depending on the application and the TCB.

[16] See also [GW14] for details on the links between these two style of embeddings.

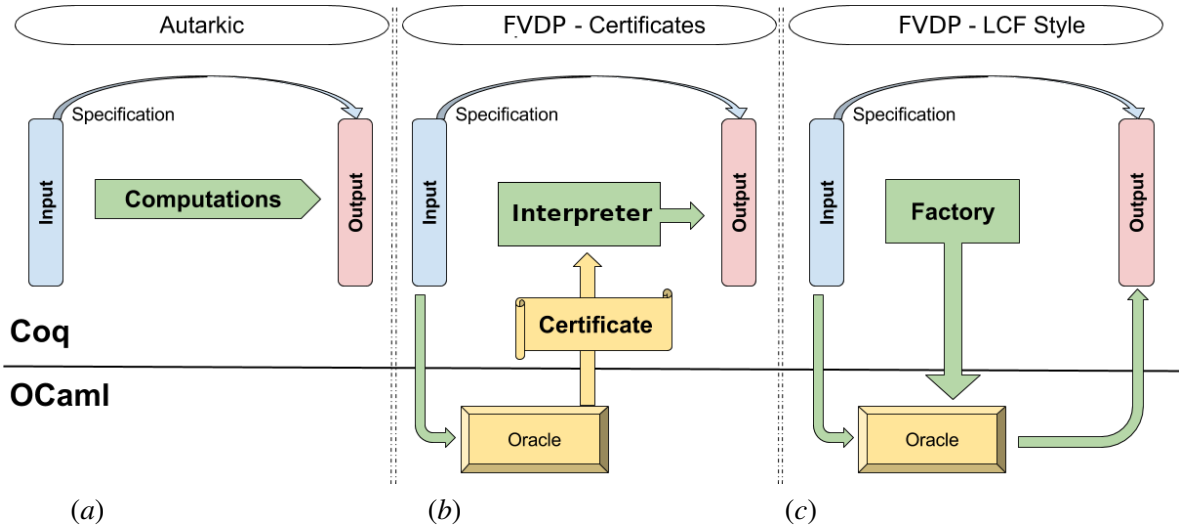


Figure 1.2: Three styles of Coq-verified computations: autarkic, certificate-producing and LCF.

Actually, to our knowledge, with [Mar17; #Bou+18], we related the first experiment of LCF style designs within a Coq+OCaml development. This is one of our contributions which Section 1.4.3 further details.

1.3.2 Decomposing Large Formal Developments thanks to Data Refinement

In order to decompose the complexity of formally verified programming, the pioneers Dijkstra [Dij68], Milner [Mil71], Wirth [Wir71] and Hoare [Hoa72] told us that it is often interesting to provide two representations of data structures:

- an "*abstract model*", which is simpler for *proofs*, e.g. for the *clients* of the data structure;
- a "*concrete implementation*", which is often more complex, but helps to provide an efficient implementation of the *elementary operations* on the data structure.

In this view, two representations of each *elementary operation* on the data structure should also be provided: the abstract one, which is the specification—simplifying reasoning about effective compositions of these elementary operations—and the concrete one which is the implementation. The principle is then to prove, *once for all*, that any *property observable* on an abstract program—*composing* the abstract operations—could also be *observed* on the corresponding concrete program—*accordingly composing* the concrete operations. When this property is established, we say the concrete operations "*refine*" (or "*simulate*") the abstract operations. Hoare [Hoa72] summarized the interest of this approach as:

> *If the data representation [also called "the data refinement"] is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program. Since abstract programs are usually very much shorter and easier to prove correct, the total task of proof has been considerably lightened by factorizing it in this way.*

In practice, given a particular *semantics* of "*abstract programs*" and "*concrete programs*" and their related notion of "*operation compositions*", given a notion of "*observable property*" (and a related notion of "*correctness*"), data-refinement methods attempt to reduce "*refinement proofs*" to simple proof

obligations. For example, Gardiner and Morgan [GM93] propose a method where data refinement helps to establish the total correctness of modular imperative programs; de Roever and Engelhardt [dRE98] compare various data-refinements methods of such programs for partial or total correctness. Because, in complex cases, it is interesting to decompose such a refinement proofs to several steps, methods like the B-method [Abr96] define an expressive specification language where a "concrete program" can itself be given as the "abstract specification" of another one. Such a composition of data refinements is called *stepwise refinement*, as coined by Wirth [Wir71].

Actually, the idea of data refinement is so fundamental that it is ubiquitous in large formal developments. But, because it needs to be specialized for some given semantics of programs and for some class of correctness properties, it is not always named "data refinement". For example, the formal correctness of a compiler can be viewed as a form of refinement: the program generated by the compiler is expected to satisfy all properties observable on the input program. In this view, the generated program is a "concrete program" and the input program is a "abstract program". Hence, the design of CompCert corresponds to stepwise data refinement: each compiler pass between two IRs (Intermediate Representations) is a data-refinement step, where "data" correspond to IR states, and where "elementary operations" are IR instructions. Indeed, Leroy [Ler09a, Def. 3, Sect 2.1] defines *safe backward simulations*, a specialized notion of data refinement, which formalizes the correctness of CompCert passes. Leroy also proposes several *simulation diagrams* which each corresponds to a particular kind of data-refinement proof [Ler09a, Sect. 3.7]. Representing data-refinement proofs by *commutative diagrams* dates back to Milner [Mil71]. See also [dRE98] for a comparative study of various kinds of commutative diagrams for data refinement.

Another very common instance of data refinement is *abstract datatype*. Indeed, it consists in abstracting a data-structure satisfying a given *representation invariant* over a given *interface* of operations. The implementations of operations are expected to preserve the representation invariant. And, the type abstraction discipline within *clients* is expected to forbid them to violate the representation invariant. For example, the "factory" of Figure 1.2(*c*) provides such an abstract datatype, which is invoked by the "oracle"—i.e. the OCaml client—in order to build objects of this datatype. This design provides a lightweight approach to ensure that the objects built by the oracle satisfy the representation invariant of the factory. Hence, it may not be a pure coincidence that an inventor of LCF style proofs [Mil79] and a pioneer of data refinement [Mil71]—a few years earlier—are the same Robin Milner (Turing Award 1991).

This document provides two other examples of design by data refinement in order to simplify large FVDP developments. This contribution is detailed in Section 1.4.4.

1.4 Contributions of this Document

Studying how to use untrusted code for producing trustworthy results dates back to de Bruijn [dBru68] and Gordon, Milner, and Wadsworth [GMW79]. It revitalized at the end of the 90's, on one side with the idea to invoke external untrusted software for helping skeptical provers [HT98], and on the other side, by "*Proof-Carrying Code*" (PCC) [NL96; Nec97; App01; App+03]: an approach where an OS kernel checks untrusted applications with the help of a formal proof that accompanies the application's executable code. Hence, PCC introduces the idea of "*certifying compilers*" [NL98], where the compiler is itself an untrusted oracle which produces both an executable and a proof of correctness. Then, Tristan and Leroy [TL08] experimented with the use of untrusted oracles for developing the CompCert *certified* compiler.[17] Since these seminal works, many other research papers

[17] There is a strong connection between "*certified*" and "*certifying*" compilers as discussed in [Ler09a, Sect. 2.2.3].

integrate untrusted computations within formal proofs. However, despite all these efforts, there are still only a few industrial-strength softwares that use formal proofs. Actually, formal verification remains a very difficult and time-consuming task.

"Standing on the shoulders of these giants", I propose to further integrate untrusted (but typesafe) OCaml code within Coq-verified code through FVDP designs. This integration is here realized by introducing a Coq library prototype, called {Impure}, and detailed in Chapter 2. The use of this library is evaluated on several "realistic" applications: Boolean SAT-solving, instruction scheduling for CompCert and linear arithmetic for static analyzers. They all involve some untrusted solver that implements advanced algorithms (i.e. CDCL for SAT-solving problems, Simplex for linear-programming problems, etc). Their FDVP design often results from a trade-off between simplicity and efficiency both in the verified component and in the untrusted solver. Choosing a "good" design may become particularly delicate when the solver needs to be instrumented in other to produce witnesses that drive the formally verified component for computing the results. For example, as we discuss below, a verbose certificate format may make the verified component simple and efficient, but might require cumbersome instrumentations of the solver and might hinder the whole computation because of a too large memory footprint.

1.4.1 Collection of FVDP Designs

Hence, this document provides an interesting bestiary of FVDP designs:

- Certifying the model found by a Boolean SAT-solvers is very simple, without any additional help from the SAT-solver (See Chap. 5).
- Certifying instruction schedulings does not require any additional witness from the untrusted solver, but require implementing nontrivial techniques in the formally verified checker, like hash-consing. Fortunately, in this case, the core of the hash-consing mechanism can be delegated to an untrusted oracle and dynamically verified thanks to a trusted pointer equality. This is an example of *iterated* FVDP design (see Chap. 3).
- For certifying "UNSAT" answers, the SAT-solver needs to produce a witness that helps the verified checker (otherwise, the untrusted solver is useless). Here, a too-naive certificate format, like the one sketched at Section 2.4.2, induces too-intrusive instrumentation on the solver [HJW14]. Fortunately, the SAT-solving community solved this issue by adopting two certificate formats: the first one—called DRAT—is easily generated by state-of-the-art SAT-solvers; the other—called LRAT—is quite easy to validate. The conversion from DRAT to LRAT is heavyweight, but delegated to an independent untrusted tool. Thus, the effort of generating an optimized LRAT certificate is factored within a single program. Our contribution here is to show how to implement an efficient Coq-verified LRAT checker, with a modest development effort. A previous attempt using Coq by Cruz-Filipe et al. [Cru+] did not scale on very large LRAT proofs. Chapter 5 details these ideas.
- Often, our formally verified code invokes some "theorems for free" deduced from the polymorphic OCaml type of the untrusted oracle. My simplest example of this technique occurs on a fast inclusion test between two lists (Section 2.4.1). The membership test to the "big" list is delegated to an untrusted oracle, which efficiently implements it thanks to a hash-table. Here, OCaml typing is powerful enough to *statically* ensure that this oracle only stores elements of the "big" list. Such an invariant deduced from OCaml typing is useful to alleviate the development effort while enabling efficient implementations, e.g. in a (polymorphic) LCF style. Actually, polymorphic LCF style helped us to implement our efficient LRAT checker with a modest development effort.

- The core of our certified computations in linear arithmetic are also driven by OCaml oracles computing over convex polyhedra. Here again, in a (polymorphic) LCF style, we exploit the fact these oracles satisfy a "theorem for free" expressing that they compute polyhedra which are—by construction—geometrically-included in their input polyhedra. See Chap. 4.
- Our certified linear overapproximation of polynomial tests invokes an untrusted oracle which drives the overapproximation according to various correct-by-construction strategies. Here the oracle provides its hints to the certified code in the form of polynomial terms enriched with ad-hoc annotations. This is also another example of *iterated* FVDP design (based on the FVDP of polyhedral computations, described above). See Section 7.3.2.

For soundness, our imperative OCaml oracles cannot be modeled as pure Coq functions. We propose here to abstract them as nondeterministic computations (within a dedicated monad). This model only induces a little burden in proofs, because we simplify proof obligations thanks to a weakest-precondition calculus. Still, when composing complex Coq computations that embed external oracles (like for our *iterated* FVDP designs of Chap. 3 and of Chap. 7), it is interesting to cleanly separate reasonings on pure data structures and algorithms from reasonings on sequences of impure computations. We achieve this using data refinement.

1.4.2 Summary of the Methodological Contributions

In summary, the main contributions of this thesis are the following:

1. the {Impure} library for modeling in Coq external untrusted OCaml oracles as nondeterministic computations;
2. a bestiary of "realistic" FVDP case studies, based on {Impure};
3. a "theorems-for-free" technique which makes FDVP even more lightweight: in particular, it leads to a design pattern for certificate-producing oracles, called "*Polymorphic* LCF *Style*" (contribution further detailed in Section 1.4.3);
4. data-refinement techniques helping for large FVDP developments (contribution further detailed in Section 1.4.4);
5. a technique to reuse polymorphic LCF style oracles within Coq tactics, independently of their embedding through {Impure}: this both reduces the TCB and may help in interactive Coq proofs. See Chap. 6.

1.4.3 FVDP by Polymorphic LCF Style and Theorems for Free

In [#Bou+18], we have introduced a design pattern, called *polymorphic* LCF *style* or *polymorphic factory style* (abbreviated in PFS), which generalizes LCF style and makes it even more lightweight. This style reduces the development effort in Coq by delegating part of the verification to the OCaml typechecker: it relies on ML polymorphism in order to ensure that invariants proved on the Coq side are preserved by imperative features of the OCaml code. This technique can be viewed as an adaptation to imperative ML of the "Theorems for Free" technique, promoted by Wadler [Wad89] relying on the parametricity of polymorphism [Rey83].

Moreover, by using polymorphism to abstract types of "theorems" instead of standard abstract datatypes, PFS is more powerful than standard LCF style: PFS oracles are simpler to design, easier to debug, more efficient and more versatile. Chapters 4, 5 and 6 argue for these claims. Actually, Chapter 6 illustrates that PFS is even useful for implementing certificate-producing style oracles pictured in Figure 1.2(*b*).

1.4.4 FVDP of {Impure} Data Structures with Data Refinement

As introduced above, iterated FVDP designs—like those of Chap. 3 and 7—require to combine within a Coq-verified computation, many "elementary" Coq-verified computations themselves embedding untrusted oracles. For the formal proofs, it is convenient to introduce a model of these "elementary" computations which is computationally pure. The bureaucracy induced by the handling of {Impure} computations is then isolated and may be quasi-automatically discharged with the help of a dedicated tactic (see Section 2.2.1).

Such a model-based decomposition of the development actually corresponds to a data-refinement design, as introduced in Section 1.3.2. We propose two data-refinement designs. The first one, detailed in Chap. 3, consists in introducing a pure model of a hash-consed data structure. Here, while the data-refinement relation (i.e. "`smem_models`" of Fig. 3.9 page 65) and the elementary operations of the data-structure (given at Fig. 3.10 of page 67) are formally defined, this example applies data refinement at an intuitive level: the example itself is not presented as an instance of a formal data-refinement framework.

On the contrary, the second one proposes a refinement framework dedicated to the FVDP of (some) abstract interpretations. The principle of abstract interpretation [CC77] is to approximate the data (or the states) of programs in order to automate the proof of some properties (e.g. absence of memory overflows) about these programs. In practice, "elementary operations" of the input program on "*concrete data*" are interpreted using an *abstract domain* approximating these operations by operations on "*abstract data*" (which enable the implementation of the targeted static analysis). Actually, such an abstract domain that relates an abstract and a concrete data representation—for a given interface of elementary operations—exactly corresponds to a data refinement. However, in abstract interpretation, the data-refinement relation is rather usually called the *concretization* relation.[18] Founded on this simple remark, our framework adapts the stepwise refinement calculus developed by Back and von Wright [BvW98] and Morgan [Mor94], and others (after [Wir71]), in order to help the formal-verification of abstract interpretations. Our refinement calculus hides the implementation details of the abstract domain, and in particular the fact that "elementary operations" of the abstract domain are impure (i.e. they may themselves invoke untrusted oracles). The use of this framework is both illustrated on a toy analyzer and a "realistic" procedure approximating polynomial computations within the {VPL} abstract domain—a certified abstract domain of convex polyhedra. See Chap. 7 for details.

Let us remark that, since [Dij68] and [Wir71], stepwise refinement is often presented in a top-down style where programs are derived from specifications. Such a top-down approach has the pedagogical interest to illustrate how an implementation and its proof are codesigned. However, in my experience, large software is developed with an agile combination of top-down and bottom-up style, guided by intuition and practice. As a consequence, I do not consider the debate "top-down vs bottom-up" as an important issue.

1.5 Quick Overview of Other Chapters

Chapter 2 introduces the {Impure} library and its "theorems-for-free". It also gives basic applications of the library with a first basic example of PFS design (a naive refutation prover on Boolean

[18] The concretization is very often defined as the upper adjoint of some monotone Galois connection. In this case, the corresponding lower adjoint is called the *abstraction*. On other abstract domains, like the one of convex polyhedra [CH78], there is no such Galois connection, but only a concretization relation, which, in this example, simply corresponds to interpret a polyhedron as a set of states.

formulas).

Chapter 3 presents a first "realistic" application of FVDP with {Impure}: instruction scheduling in {CompCert-KVX}, with an interesting use of OCaml pointer equality.

Chapter 4 is an extended tutorial to PFS designs with {Impure}, illustrated on two "realistic" operators of the {VPL} abstract domain.

Chapter 5 demonstrates with {SatAnsCert} that such PFS designs provide a lightweight but efficient technique for Coq-verified SAT-solving.

Chapter 6 details how the {VplTactic} Coq-plugin invokes a PFS oracle of the {VPL}, and why this is useful.

Chapter 7 describes our refinement calculus for FVDP of abstract computations with the {VPL}.

Chapter 2

Toward Lightweight FVDP by Combining Coq and ML Typechecking[†]

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus [Rey83].

Philip Wadler in "*Theorems for Free!*" [Wad89].

Eventually I came to regard nondeterminacy as the normal situation, determinacy being reduced to a [...] special case.

Edsger W. Dijkstra in "*A Discipline of Programming*" [Dij76].

Contents

[†]Fragments of this chapter have been published in [#FB14; #Bou+18; #SBM20].

Section 2.1 illustrates why the standard embedding of OCaml oracles into Coq-verified computations through extraction—as used in CompCert—is unsound. Section 2.2 presents an alternative embedding—called the {Impure} FFI (Foreign Function Interface)—and discusses its soundness. It relies on a parametricity property of the underlying "Coq+OCaml" type system, which provides powerful "theorems for free" about OCaml polymorphic oracles. These "theorems for free" can be viewed as a relaxation of Wadler's ones in presence of Dijkstra's nondeterminacy[1]. Section 2.3 applies this parametricity property to extend the Coq programming language with some polymorphic impure operators: exception-handling and fixpoints. This parametricity property is also the basis of a design pattern called "Polymorphic Factory Style", alleviating FVDP of certificate-producing oracles, and introduced by Section 2.4. At last, Section 2.5 presents caveats and future works about the {Impure} FFI.

2.1 Unsoundness of the Standard FFI w.r.t OCaml

The register allocation of CompCert is declared in Coq by[2]

```
Parameter regalloc: RTL.function → res LTL.function.
```

Here, "`Parameter`" is synonymous with "`Axiom`" and "`res`" is quite similar to the "`option`" type transformer. A Coq directive in CompCert instructs Coq extraction [Let08] to replace axiom "`regalloc`" by a function of the same name from the `Regalloc` OCaml module. While very common, this approach is potentially unsound.

Let us consider the Coq example on the right-hand side. It first defines a constant `one` as the Peano's natural number representing 1. Then, it declares an axiom `test` replaced at extraction by a function `oracle`. Finally, a lemma `cong` is proved, using the fact that `test` is a function.

```
Definition one: nat := (S O).
Axiom test: nat → bool.
Extract Constant test ⇒ "oracle".

Lemma cong: test one = test (S O).
  auto.
Qed.
```

However, implementing `oracle` by "`let oracle x = (x == one)`" in OCaml makes the lemma `cong` false at runtime. Indeed, `(oracle one)` returns **`true`** whereas `(oracle (S O))` returns **`false`**, because "==" tests equality between *pointers*. Hence, the Coq axiom is unsound w.r.t this implementation. A similar unsoundness is obtained with another implementation of `oracle`, that returns the value of a global reference, containing **`true`** at the first call, and **`false`** at the second call:

```
let oracle = let h=ref false in (fun x -> h:=not !h; !h)
```

This unsoundness comes from the fact that a Coq function f is *pure*: it satisfies "$\forall x, (f\ x) = (f\ x)$", whereas an OCaml function may not satisfy this property.

In CompCert, the implementation of `regalloc` uses mutable data structures[3]: it is thus not obvious whether `regalloc` is observationally pure or not – in the Coq sense. But, the implicit assumption

[1] The remainder of this document uses "nondeterminism" instead of "nondeterminacy" (because it seems more widespread)

[2] See https://github.com/AbsInt/CompCert/blob/master/backend/Allocation.v

[3] See https://github.com/AbsInt/CompCert/blob/master/backend/IRC.ml

about `regalloc` purity, derivable from its declaration in Coq, should not hide a critical issue, even in case of unexpected bug in `regalloc`. Indeed, the remainder of the compiler never compares executions of `regalloc`: it does not depend on whether `regalloc` is pure or not. In other words, this implicit assumption only results from a shortcut in the formalization and is expected to be useless, without any bad consequence if it is wrong.[4] Nevertheless, avoiding such a useless assumption would help to promote a more systematic use of oracles.

2.2 {Impure}: a Coq Library to Import ML Foreign Functions

Fouilhé and Boulmé [#FB14] have proposed to avoid this unsoundness by modeling external OCaml functions using a notion of nondeterministic computations. With respect to the previous example, if the result of `test` is declared to be nondeterministic, then the property `cong` is no longer provable. For a given type A, type $??A$ represents the type of nondeterministic computations returning values of type A: it can be interpreted as $\mathcal{P}(A)$, the type $A \to \text{Prop}$ of predicates over A. Formally, the type transformer "$??$." is axiomatized as a monad that provides a *may-return* relation $\rightsquigarrow_A$ of type $??A \to A \to \text{Prop}$. Intuitively, when "$??A$" is seen as "$\mathcal{P}(A)$", then "$\rightsquigarrow$" simply corresponds to the identity function. At extraction, $??A$ is extracted like A, and its binding operator is efficiently extracted as an OCaml let-in.

For example, replacing the `test` axiom by "`Axiom test: nat → ??bool`" avoids the above unsoundness w.r.t the OCaml `oracle`. The `cong` property can still be expressed as below, but it is no longer provable – because it is not satisfied when interpreting $??A$ as $\mathcal{P}(A)$ and interpreting `test` as the function returning the trivially true predicate (in this interpretation, the goal below reduces to the false property that all Booleans are equals).

```
cong: ∀ b b', (test one)⇝b →(test (S O))⇝b' →b=b'.
```

In other words, the $A \to ??B$ represents the type of *impure functions* from type A into type B. Informally, we interpret the type $??B$ as $\mathcal{P}(B)$ the type of predicates characterizing the possible results. Thus, this interpretation represents each impure function as a function of $A \to \mathcal{P}(B)$, or equivalently, as a relation of $\mathcal{P}(A \times B)$, because of the bijection between these two types.

The {Impure} library turns Coq+OCaml into a kind of dependently type imperative programming language. In contrast to previous works [CSW14], its metatheory is not yet well established (as detailed in Sections and 2.5). Its interest is to provide a powerful (but very simple) implementation, able to combine Coq with arbitrary polymorphic imperative OCaml code. For example, [CSW14] lacks support for polymorphism, whereas this is the key feature enabling "theorems-for-free", applied in all case studies of this document.

Section 2.2.1 defines type $??B$ using axioms in order to provide an efficient extraction into OCaml, where "$??$" are simply removed. Based on this notion of impure computations, Section 2.2.2 presents the core of the {Impure} FFI. Section 2.2.3 explains how this FFI is related to a parametricity property of the underlying Coq+OCaml type system. Finally, Section 2.2.4 extends the FFI with pointer equality.

[4] In practice, this implicit assumption does not seem the main weakness of CompCert, as discussed in Section 1.2.5.

2.2.1 Definition of the May-Return Monad in the {Impure} library

This section introduces in an informal syntax the theory of the may-return monads and presents the informal interpretation of these axioms. See the sources online[5] for the full Coq syntax with the proofs. The definition of may-return monads in this document – given below – is inspired by the original definition of [#FB14], itself inspired by the structure of monads in functional programming languages [Wad95]. There are however two differences between the definition below and the original one. First, in this document, the congruence "≡" over computations has been omitted. Indeed, in the {VPL}, the Verified Polyhedra Library of [#FB14], this congruence is only needed in order to prove a property on a higher-order operator that is absent in the case studies of this document. Moreover, as discussed in Section 2.5.3, the meaning of such an equality with respect to the extracted code is counterintuitive: an issue that we keep out of the scope of this document. Second, this document introduces the "mk_annot$_A$" operator, necessary to derive "theorems for free" on higher-order operators.

Definition 2.1 (May-return monad)**.** For any type A, type $??A$ represents impure computations returning values of type A, and provides a may-return relation

$$\rightsquigarrow_A : ??A \to A \to \text{Prop}$$

where "$k \rightsquigarrow a$" means that "*k may return a*". It also provides the three following operators

- Operator $\gg\!=_{A,B} : ??A \to (A \to ??B) \to ??B$ encodes an impure OCaml "**let** $x = k_1$ **in** k_2" into "$k_1 \gg\!= \lambda x, k_2$". This operator must satisfy

$$k_1 \gg\!= k_2 \rightsquigarrow b \;\to\; \exists a, k_1 \rightsquigarrow a \wedge k_2\, a \rightsquigarrow b$$

- Operator $\varepsilon_A : A \to ??A$ lifts a pure value as an impure computation. It must satisfy

$$\varepsilon\, a_1 \rightsquigarrow a_2 \;\to\; a_1 = a_2$$

- Operator mk_annot$_A : \forall (k : ??A),\ ??\{a \mid k \rightsquigarrow a\}$ annotates the result of a computation k with an assertion expressing that it has been returned by k. This operator is extracted as identity (it has no computational contents). Its only purpose is to embed $\rightsquigarrow$ into Coq dependent types.[6] For example, given $k_1 : ??A$ and $k_2 : (??\{a \mid k_1 \rightsquigarrow a\} \to ??B)$, then mk_annot$k_1 \gg\!= k_2 : ??B$. In practice, such an operator is needed for proving properties of higher-order operators by parametricity: an example will be detailed in Figure 2.7.

In the Coq code, "$k_1 \gg\!= \lambda x, k_2$" is written with a Haskell-like notation "`DO x <~ k1 ;; k2`" (or "`k1 ;; k2`" if `x` does not appear in `k2`). And ε is written "`RET`".

Interpretations of May-Return Monads

Here is the interpretation of "$??A$" as the type of predicates "$\mathcal{P}(A)$": function $\rightsquigarrow_A$ is identity on $\mathcal{P}(A)$; ε_A is the identity relation of $A \to \mathcal{P}(A)$; $\gg\!=_{A,B}$ returns the image of a predicate on A by a binary relation of $A \to \mathcal{P}(B)$; mk_annot returns the trivially true predicate. Theses definitions are formalized in Figure 2.1. They satisfy axioms of may-return monads.

Actually, it is worth noticing that usual monads are naturally embedded as a may-return monad. For example, Figure 2.2 corresponds to the embedding of the identity monad. And, Figure 2.3 corresponds to the embedding of the state-monad on a given global state of type S.

[5] http://github.com/boulme/Impure/blob/master/ImpMonads.v

[6] As noted by [CSW14, Section 3.1], monad programming is not directly compatible with dependent type programming, because the bind operator ($\gg\!=$) corresponds to a non-dependent application of function. In this context, "mk_annot$_A$" is a workaround for embedding dependent types involving monadic computations within monadic types.

$$??A \triangleq A \to \text{Prop} \qquad k \rightsquigarrow a \triangleq (k\, a)$$

$$\varepsilon\, a \triangleq \lambda x, a = x \qquad k_1 \gg\!= k_2 \triangleq \lambda x, \exists a, (k_1\, a) \wedge (k_2\, a\, x) \qquad \text{mk\_annot}\, k \triangleq \lambda x, \text{True}$$

Figure 2.1: Power-set instance of may-return monads

$$??A \triangleq A \qquad k \rightsquigarrow a \triangleq k = a$$

$$\varepsilon\, a \triangleq a \qquad k_1 \gg\!= k_2 \triangleq (k_2\, k_1) \qquad \text{mk\_annot}\, k \triangleq \text{exist}_{\rightsquigarrow}\, k\, \text{eq\_refl}_k$$

where
- $\text{exist}_{\rightsquigarrow}$ is the constructor of the dependent pair $\{\, a \mid k \rightsquigarrow a \}$
- eq_refl_k is a proof of $k = k$

Figure 2.2: Identity instance of may-return monads

$$??A \triangleq S \to A \times S \qquad k \rightsquigarrow a \triangleq \exists s, \text{fst}(k\, s) = a$$

$$\varepsilon\, a \triangleq \lambda s, (a, s) \qquad k_1 \gg\!= k_2 \triangleq \lambda s, \text{let}\ (a, s') := (k_1\, s)\ \text{in}\ (k_2\, a\, s')$$

$$\text{mk\_annot}\, k \triangleq \lambda s, \text{let}\ (a, s') := (k\, s)\ \text{in}\ (\text{exist}_{\rightsquigarrow}\, a\, p_{k,s},\ s')$$

where $p_{k,s}$ is a proof of $\exists s_0, \text{fst}(k\, s_0) = \text{fst}(k\, s)$

Figure 2.3: State-transformer instance on a global state of type S

In order to handle impure computations in Coq, the {Impure} library declares an abstract may-return monad (i.e. its implementation remains hidden).[7] It is extracted like the identity may-return monad of Figure 2.2 except that, in order to enforce the expected evaluation order, operator $\gg=$ is extracted to the reverse application operator of the OCaml standard library, written `(|>)` (and defined by "`let (|>) x f = f x`").

Reasoning on Impure Computations with Weakest-Liberal-Preconditions

Having introduced axioms for impure computations in Definition 2.1, we automate Coq reasoning about such computations, by reusing a weakest-precondition calculus introduced by [#FB14]—itself inspired by [Dij75]— and programmed as a very simple Ltac tactic, called `wlp_simplify`.

This calculus relies on a transformation of {Impure} computations into computations over predicates, defined in Coq by:

$$\text{wlp}_A : ??A \to (A \to \text{Prop}) \to \text{Prop} \qquad \text{such that} \qquad \text{wlp}\ k\ P \triangleq \forall a, k \rightsquigarrow a \to (P\ a).$$

In other words, (wlp k P) expresses the weakest (liberal) precondition ensuring that any result returned by computation k satisfies postcondition P. When $??A$ is interpreted as $\mathcal{P}(A)$, wlp corresponds to inclusion of predicates. Now, we define the notion of WLP-theorems.

Definition 2.2 (WLP-theorems)**.** A WLP-theorem is a Coq theorem with a conclusion of the form "(wlp k P)". Such a theorem means that (under the parameters of the theorem),

For all r, if the extraction of k returns the extraction of r, then r satisfies P.

In particular, when the extraction of k does not terminate or raises an uncaught exception, WLP-theorems do not give any useful information (as usual in *partial correctness*). In our Coq code, we write (wlp f $\lambda r, P$) with the notation "`WHEN f ⇝ r THEN P`". For example, let us consider the following Coq code:

```
Variable f: nat → ?? nat.
Definition g (x:nat): ?? nat := DO r ↜ f x;; RET (r+1).
Lemma triv: ∀ x, WHEN g x ⇝ r THEN r > 0.
```

The `wlp_simplify` tactic simplifies this goal into the trivial property "$\forall n : \mathbb{N}, n+1>0$".

This tactic proceeds backward on wlp-goals, by applying repeatedly lemmas which are represented below as rules. It first tries to apply backward a *decomposition* rule: one for unit or bind below, or one for pattern-matching over some usual types (Booleans, option types, product types, etc.). When no decomposition applies, the tactic tries to apply Call and then tries to discharge the left premise using an existing lemma—chosen in a user-given hint basis by tactic `eauto`; if this succeeds, the goal is replaced using the right premise; otherwise, the Call rule is considered as having failed, and the definition of wlp is simply unfolded (this typically suffices for dealing with mk_annot).

$$\textsc{Decomp-unit}\ \frac{P\ a}{\text{wlp}\ (\varepsilon\ a)\ P} \qquad \textsc{Decomp-bind}\ \frac{\text{wlp}\ k_1\ \lambda a.(\text{wlp}\ (k_2\ a)\ P)}{\text{wlp}\ (k_1 \gg= k_2)\ P}$$

$$\textsc{Call}\ \frac{\text{wlp}\ k\ P_1 \qquad \forall a, k \rightsquigarrow a \to (P_1\ a \to P_2\ a)}{\text{wlp}\ k\ P_2}$$

Hence, the `wlp_simplify` tactic automates the decomposition of impure computations, while injecting existing lemmas about impure functions through the Call rule. Indeed, the Call rule is typically

[7] See http://github.com/boulme/Impure/blob/master/ImpConfig.v

applied when k is of the form "$f\ t_1\ \ldots\ t_n$"—where f an impure function—and the "wlp $k\ P_1$" premise is discharged by applying a WLP-theorem about f. In this case, the original goal "wlp $k\ P_2$" is reduced to a proof of "$P_1 \Rightarrow P_2$" (hypothesis "$k \rightsquigarrow a$" being only here for helping the user to recover the origin of the proof obligation). But, the CALL rule requires a strict discipline in order to be useful:

1. it often requires f to be an opaque constant (otherwise, some decomposition rule depending on the implementation of f applies instead of the CALL rule);
2. there must be only one WLP-theorem about f in the hint basis;
3. the parameters of the WLP-theorem should only bind the parameters of f: extra parameters or extra hypotheses (e.g. on the parameters of f) should be bound in the postcondition P.

For example, the `exp_hterm_correct` theorem about `exp_hterm` in Figure 3.10 (page 67) has exactly the same parameters `e`, `hd` and `hod` as `exp_hterm`. The "hypotheses" about these parameters and the extra-parameters `ge`, `od`, `d` and `m` invoked in these hypotheses are actually bound in the postcondition of the WLP-theorem that appears after the "`THEN`" keyword. Nested WLP-formulas are supported and useful for higher-order operators, such as theorem `hCons_correct` of page 61, that specifies a higher-order memoizing factory called `hCons`.

The constraint of using at most one WLP-theorem (typically called f`_correct`) for each impure function f does not seem too restrictive in practice. It is naturally satisfied in developments of impure functions by data refinement, where each impure function is modeled with a pure function or a pure relation. For example, the `exp_hterm_correct` theorem in Figure 3.10 expresses that if `hd` is modeled by `d` and `hod` is modeled by `od` then, the result `ht` of `(exp_hterm e hd hod)` is modeled by the pure `(exp_term e d od)`. In the rare cases where it is not convenient to specify some impure function with a single theorem, we may still introduce several hint bases in order to control how to solve the left-premises of CALL rules.

In our Coq development, `wlp_simplify` automates many bureaucratic reasonings on the monad operators. This very simple tactic could probably be improved to manage pattern-matching more systematically, and to provide user control of the naming of intermediate hypotheses (for example, by interacting with the "LibHyps" library of Pierre Courtieu[8]).

2.2.2 The Core of the Foreign-Function Interface (FFI) Provided by {IMPURE}

As shown in introduction, declaring external OCAML oracles in COQ may be *unsound*, by authorizing COQ theorems that can be false at runtime. The may-return monad was introduced in order to avoid the pitfall of embedding impure computations as pure functions. But this is not sufficient to ensure soundness. To this end, we need to define a class "*permissive*" of COQ types and a class "*safe*" of OCAML values satisfying Hypothesis 1 below, with "*being permissive*" and "*being safe*" automatically checkable, and as expressive as possible. In this document, we consider the following definition for "*safe*". The definition of "*permissive*" will be gradually introduced up to Definition 2.4.

Definition 2.3 (Safe OCAML value)**.** An OCAML value is "*safe*" **iff** it is well-typed and is not a closure invoking external polymorphic functions such as `Obj.magic`, and without using cyclic values *of a type extracted from* COQ, such as "`let rec v = S v`".[9]

NB: The last restriction does not forbid safe OCAML values to contain recursive functions nor cyclic mutable data structures.

Hypothesis 1 (Soundess of permissive COQ types)**.** Every "*permissive*" COQ type T according to Definition 2.4 satisfies the following property:

[8] https://github.com/Matafou/LibHyps

[9] Section 2.5.1 details issues of cyclic values on COQ extracted types.

Every safe OCaml *value compatible with the extraction of T is "*soundly*" axiomatized in* Coq *with type T – in the sense that WLP-theorems deduced from the axiom cannot be falsified when running the extracted code, in which the axiom has been replaced by the* OCaml *value.*

Ideally, we aim to extend Coq with a "`Import Constant`" construct of the form:

```
Import Constant ident: permissive_type := "safe_ocaml_value".
```

and acting like "`Axiom ident: permissive_type`", but with additional checks during Coq and OCaml typechecking in order to ensure soundness of extraction. However, defining precisely such typechecking algorithms is left for future work.

Definition of Permissivity

This section gradually introduces our definition of *permissivity* in order to explain Hypothesis 1. We first give examples of unsound types, and examples that are conjectured to be sound. Section 2.1 have illustrated that type `nat→bool` is unsound: thus, it cannot be permissive. On the contrary, type `nat→??bool` is conjectured to be sound. We also conjecture that `nat→??nat` is sound. But, type `nat→??{n:nat | n ≤ 10}` – extracted to "nat->nat" – is not. Indeed, such a Coq type corresponds to assume a *postcondition* on the oracle that the OCaml typechecker cannot ensure.

Similarly, type `nat→??(nat→nat)` – extracted to "`nat->(nat->nat)`" – is unsound because the Coq side expresses that the result of type `nat→nat` is pure, whereas this (implicit) postcondition cannot be ensured by OCaml typechecker. Actually, the same phenomenon happens with `nat→(nat→??nat)` (extracted on the same OCaml type): the partial application on the first argument is declared pure in Coq, whereas this cannot be ensured by OCaml typechecker.

In contrast, types `nat→??(nat→??nat)` and `(nat→??nat)→??nat` are conjectured to be sound. And also `{n | n ≤ 10}→??nat`. On this last example, the Coq axiom requires a *precondition* that OCaml typechecker can safely ignore. A similar phenomenon happens with `(nat→nat)→??nat`: the purity of the parameter is an implicit precondition that OCaml typechecker can safely ignore. Note that currying—like in `nat→??(nat→??nat)`—allows for more imperative OCaml implementations (at the price of more bureaucracy on the Coq side) than uncurrying—like in `nat*nat→??nat`.

In the general case, permissivity can be viewed as a given *supertyping* relation between Coq types and OCaml types: a Coq type is permissive only if it is a supertype of its extraction. In this view, permissivity of arrow types requires to distinguish "inputs" (negative occurrences) from "outputs" (positive occurrences): outputs are covariant and inputs are contravariant. We thus also need to introduce "*permissible* Coq *types*", i.e. Coq types that are a subtype of their extraction.

Definition 2.4 (Permissive & Permissible Coq types)**.** Permissive and permissible Coq types are defined by mutual induction:

permissible types (i.e. Coq types allowed in inputs of oracles)

- an inductive type is permissible whenever its sort is **`Prop`**, or whenever the type of each input of each constructor is permissible (or the inductive type under definition). For example, `nat` and `{n:nat | n ≤ 10}` are permissible.
- a forall type, or an arrow type, or a type of the form "`_ → ?? _`", is permissible whenever its input type is permissive and its output type is permissible.

permissive types (i.e. Coq types allowed in outputs of oracles)

- an inductive type is permissive whenever its sort is not **Prop** and whenever the type of each input of each constructor is permissive. See example below.
- an arrow type is permissive whenever the arrow is followed by a "??", and its input type is permissible, and its output type is permissive.
- ML polymorphism – i.e. prenex universal polymorphism – preserves permissivity.

For example, given type `foo` below, type `nat→??foo` is permissive. But this would not be the case if constructor `Bar` has no "??" in the type of its argument.

```
Inductive foo := Bar: (nat → ??nat) → foo
```

Remark that some types such as "`(nat → nat) → nat`" are neither permissive nor permissive: they cannot be output types nor input types of oracles.

A more advanced example of permissive type is given by the polymorphic type of `make_cref` in Figure 2.4. Section 2.2.3 illustrates that Hypothesis 1 implies a powerful parametricity property on such a polymorphic oracle.

```
Record cref {A} := { set: A → ?? unit; get: unit → ?? A }.
Axiom make_cref: ∀ {A}, A → ?? cref A.
```

Figure 2.4: A Coq FFI of mutable references

```
let make_cref x =
  let r = ref x in { set = (fun y -> r:=y); get = (fun () -> !r) }
```

Figure 2.5: Standard OCaml implementation of `make_cref`

```
let make_cref x =
  let h = ref [x] in {
    set = (fun y -> h:=y::!h);
    get = (fun () -> List.nth !h (Random.int (List.length !h))) }
```

Figure 2.6: Iconic variant of `make_cref`

Application to Imperative Programming in Coq

Let us start exploring basic imperative programming in Coq, by using mutable data structures and I/O. Let us first consider the embedding of mutable references with the Coq code of Figure 2.4: it defines the record type `cref` that represents references in a kind of object-oriented style (as the pair of a mutator `set` and a selector `get`), and declares an oracle `make_cref` building values of this type. On the OCaml side, type `cref` is extracted to

```
type 'a cref = { set: 'a -> unit; get: unit -> 'a }
```

Then, we define `make_cref: 'a -> 'a cref` such that it allocates a fresh reference `r` and returns the pair of `set`/`get` function to update/access the content of `r` (see the code in Figure 2.5). Hypothesis 1 states that it is sound to implement `make_cref` by any *safe* OCaml function of type `'a -> 'a cref` like in Figure 2.5. Having implemented `make_cref` according to Figure 2.5, the user can thus program

with mutable references in Coq. However, most properties of this implementation cannot be *formally* proven in Coq.

Indeed, from the formal point-of-view, any *safe* OCaml function of type `'a -> 'a cref` is admitted as a sound implementation of `make_cref`, including the *iconic* implementation of Figure 2.6. I qualify it with "*iconic*", because this implementation typifies what any OCaml implementation of `make_cref` can do: store inputs of `make_cref` and `set` such that `get` outputs one of the previously stored inputs. For example, every execution using the implementation of Figure 2.5 can be emulated by an execution using the implementation of Figure 2.6 where each call to `Random.int` returns 0: in this way, `get` outputs the last received input.

Hence, all formal properties provable from the interface of Figure 2.4 should be satisfied by the oracle of Figure 2.6. Thus, they can only express that if all the inputs of a given reference satisfy some given *invariant*, then the value returned by `get` will also satisfy this invariant. Such a property can be partly expressed in Coq by instantiating the parameter `A` of `cref` in Figure 2.4 on a Σ-type that constrains this reference to preserve the given invariant. Whereas this technique seems a bit weak on this example, Section 2.3.3, Section 2.4.1 and Chapter 5 present interesting applications of this lightweight technique for constraining polymorphic mutable data structures (such as hash-tables). Finally, let us note that our embedding of ML references does not forbid aliases as long as they are compatible with Coq typing: see details in Section 2.5.2.

However, extending extracted code with an OCaml main could in theory break some properties proved on the Coq side (see also Section 2.5.2). It is thus safer to define the main function of executables on the Coq side. This motivates embedding some I/O functions in Coq. Such an embedding is very easy. Currently, the {Impure} library provides a few wrappers of some functions of the OCaml standard library, such as these two (where `pstring` is a Coq type to represent strings).

```
Axiom read_line: unit → ?? pstring.   (* reads a line from stdin *)
Axiom println: pstring → ?? unit.     (* prints a line on stdout *)
```

However, the {Impure} library does not provide any formal reasoning support on these I/O functions. Hence, in this approach, reasoning with I/O on Coq code remains informal – more or less like on OCaml code. The programmer is only much more protected against stupid mistakes when combining formally proved code and trusted (but informally verified) code, because the Coq type system is more accurate.[10]

2.2.3 Parametricity by Invariants (i.e. "Theorems for Free" about Oracles)

According to Definition 2.4, a polymorphic type such as "`∀ A, A→??A`" is permissive. Together with Hypothesis 1, this implies a "theorem for free" on *safe* OCaml values of the corresponding extracted type. For example, we now prove that any `pid`, defined as a safe OCaml value of type `'a -> 'a`, satisfies "*when* `(pid x)` *returns normally some* `y` *then* `y = x`".

In the following, we say that a function `pid` satisfying the above property is a *pseudo-identity* (indeed, it may not be the identity because it may not terminate normally or produce side-effects).

In order to prove that any safe "`pid:'a -> 'a`" is a pseudo-identity, we first declare `pid` as an external function in Coq. Then, we build a Coq function `cpid`, which is proved to be a pseudo-identity, and which is extracted to OCaml like "`let cpid x = (let z = pid x in z)`". In the Coq source, for a type `B` and a value `x:B`, `(cpid x)` invokes `pid` on the type `{y:B|y=x}`, which constrains it to produce a value that is equal to `x`. Below, `‘z` returns the first component of the dependent pair `z` of

[10] This contrasts with FreeSpec [LR20] or InteractionTrees [Xia+20], which allow for reasoning on side-effects within Coq, but do not seem to enable the use of polymorphic oracles.

type {y:B|y=x}; the **Program** environment allows for terms with "holes" (like here in the implicit coercion of x:B into a value of {y:B|y=x}) and generates static proof obligations to fill the holes.[11]

```
Axiom pid: ∀ A, A→??A.
Program Definition cpid{B}(x:B):??B := DO z ↞ pid {y|y=x} x;; RET ‘z.
Lemma cpid_correct A (x y:A): WHEN (cpid x) ⇝ y THEN y=x.
```

Let us point out that we cannot prove in Coq that pid – declared as the axiom given above – is a pseudo-identity. Indeed, we provide a model of this axiom where pid detects – through some dynamic typing operators – if its parameter x has a given type Integer and in this case returns a constant value, or otherwise returns x. Such a counterexample already appears in [VW07]. This function is now provided in Java syntax.

```
static <A> A pid(A x) {
  if (x instanceof Integer) // A==Integer, because Integer is final
    return (A)(new Integer(0));
  return x;
}
```

The soundness of cpid extraction is thus related to a nice feature of ML: type erasure in ML semantics ensures that *type-safe* functions handle polymorphic values in a uniform way.

Actually, a similar counterexample is now given by the following *unsafe* OCaml function, which exploits the low-level representation of integers, and shortcuts the type checker with Obj.magic.

```
let pid (x:'a) : 'a =
  if Obj.is_int (Obj.repr x) then Obj.magic 0 else x
```

This explains why such a counterexample must be rejected by Definition 2.3.

In summary, our Coq proof is not about pid, but about cpid which instantiates pid on a dependent type. Actually, cpid and pid coincide, but only *in the extracted code*. This proof can be viewed as a "theorem for free" in the sense of Wadler [Wad89]: it is a parametricity proof for a *unary* relation, i.e. a predicate that we call here an *invariant*. Bernardy and Moulin [BM12; BM13] have previously demonstrated that parametricity reasoning can be constructively internalized in the logic from an erasure mechanism. Here, in our "Coq+OCaml" logic of programs, it is associated to the fact that the invariant instantiating the polymorphic type variable in the Coq proof is syntactically removed by Coq extraction.

But, whereas parametricity of pure System F has been established a long time ago by Reynolds [Rey83], its adaptation to imperative languages with higher-order references *à la* ML is much more recent [ADR09; DAB09; Bir+11]. Indeed, because higher-order references allows building recursive functions without explicit recursion (see Figure 2.9 page 37), it is even hard to define what is a predicate over such a higher-order reference. See [AAV02; App+07; HDA10] and [App14, Part V]. This document leaves the proof of Hypothesis 1 for future works, and focuses on demonstrating its powerful applications.

In another line of work, Keller and Lasson [KL12] followed by Anand and Morrisett [AM17] have proposed an internalization of parametricity within Coq, but in a very different way and for a very different purpose than {Impure}. Their parametricity reasoning applies over *binary* relations on *pure* Coq terms. Their goal is to transfer "for free" properties which are proved with one data representation into another one. This suggests that we may view parametricity as "data-refinement proofs for free".

[11] This small example also illustrates how our approach benefits from powerful features of Coq such as **Program**.

2.2.4 Axioms of the Trusted Equality of Pointers

We now extend the FFI described at Section 2.2.2, by embedding the physical equality (i.e. pointer equality) of OCaml into Coq. In contrast to all other oracles in this document, we impose the `phys_eq` oracle to satisfy an axiom – called `phys_eq_true` – in addition to its declaration. Thus, the implementation of this oracle must be trusted.

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool.
Extract Constant phys_eq ⇒ "(==)".
Axiom phys_eq_true: ∀ A (x y: A), phys_eq x y ⇝ true →x=y.
```

As illustrated on the example of Section 2.1, because "`(==)`" distinguishes pointers: it can distinguish values that the Coq logic cannot. Because Coq propositions are implicitly considered modulo structural equality of terms, they cannot speak about the underlying pointers that represent terms.

Hence, "`phys_eq x y ⇝ b`" means something like "*if* `b=true` *then it has existed an allocated object* `o` *such that* `x=o=y` *(for structural equalities)*". For example, the following property is trivially provable in our model:

```
Lemma trivial: ∀ x y, x = y →  phys_eq x x ⇝ true →  phys_eq x y ⇝ true.
```

We may also simply interpret "`phys_eq x y ⇝ b`" as the proposition "`b=true → x=y`". Hence, I claim that we cannot prove properties such as the one below, that are falsifiable by `(==)` at runtime, because those properties are also false for that simple interpretation:

```
∀ (x y:nat), x=y → phys_eq x x ⇝ true → phys_eq x y ⇝ false → False.
```

In conclusion, our "`phys_eq`" model of OCaml `(==)` does not speak about pointers: it simply expresses that `(==)` is able to establish some structural equalities (in a nondeterministic way from Coq eyes).[12] And, I will use the `phys_eq_true` axiom in order to replace some tests about structural equality by faster tests using physical equality instead. Section 2.3.3 gives an example. It also been used to implement a verified hash-consing mechanism in the CompCert backend. See Section 3.3.2.

2.3 Extending Coq "for free" with Higher-Order Impure Operators

This section applies the {Impure} FFI in order to extend the Coq programming language with some polymorphic impure operators[13]: exception-handling at Section 2.3.1, loops at Section 2.3.2 and (memoized) fixpoints at Section 2.3.3. Our goal is to formally prove the *usual rules of Hoare logic* for these operators *for partial correctness*. This is achieved by applying the technique of "*parametricity by invariants*" (introduced at Section 2.2.3): we derive these correctness rules by instantiating the polymorphic type of well-chosen oracles on a well-chosen sigma-type. In other words, we illustrate that "parametricity by invariants" interprets ML polymorphic types as "*higher-order invariants*", i.e. invariant properties (of ML values) depending on type variables which themselves names some invariant. Hence, with this interpretation, ML typecheckers are powerful engines to infer higher-order invariants for partial correctness.

Note that extending Coq with nonterminating loops is well-known (e.g. [Chl13, Chap. 7]). The novelty of my approach is to derive such loops from arbitrary oracles implementing a given ML type.

[12] Our model of OCaml pointer equality thus differs from the one of Breitner et al. [Bre+18] which represents the pointer equality of Haskell as a pure Coq function. Indeed, modeling physical equality as a pure function allows proving falsifiable theorems, as shown in Section 2.1.

[13] The full Coq/OCaml code of these examples is online at https://github.com/boulme/ImpureDemo

In particular, this is applied to a formally verified memoized fixpoint operator in Section 2.3.3. Again, such a formally verified memoized fixpoint is not new: for example, it could be done in separation logics encoded in Coq (e.g. Ynot [Nan+08; Chl+09]). The novelty of our approach is that there is no more proof effort for the memoized fixpoint than for the naive fixpoint. The only overhead is a small defensive check at runtime (i.e. physical equality test for the naive fixpoint, or the replay of the equality test involved in the memoization for the memoized fixpoint).

2.3.1 Exception-Handling Operators

First, we declare an external function `fail` which is (informally) expected to raise an error parametrized by a string.

```
Axiom fail: ∀ {A}, pstring → ?? A.
```

This axiom is safely implemented by the following OCaml function `fail: pstring -> 'a`.

```
exception ImpureFail of pstring
let fail msg = raise (ImpureFail msg)
```

But, this axiom is also safely implemented by the following OCaml function `fail: 'a -> 'b`.

```
let rec fail msg = fail msg
```

Actually, while our *formal* Coq reasonings (on the partial correctness) will be valid for any of these implementations, our *informal* reasonings (on the performance) will only consider the first implementation.

For its formal correctness, `fail` never returns a value, or equivalently it returns only values satisfying any predicate. In order, to get this property "for free", we wrap `fail` into a function `FAILWITH` of the same type, but which internally calls `fail` on the empty type `False`. For any value `r:False` returned by `fail`, we are thus able to build any value of any type (by destructing `r`).

```
Definition FAILWITH {A:Type} msg: ?? A :=
  DO r ↭ fail (A:=False) msg;; RET (match r with end).

Lemma FAILWITH_correct A msg (P: A → Prop):
  WHEN FAILWITH msg ⇝ r THEN P r.
```

Now, we use `FAILWITH` to define dynamic assertion checking. Below, "`assert_b`" ensures that a (pure) Boolean expression is `true` or aborts the computation otherwise.

```
Program Definition assert_b (b: bool) (msg: pstring): ?? b=true :=
  match b with
  | true ⇒ RET _
  | false ⇒ FAILWITH msg end.
Lemma assert_correct msg b: WHEN assert_b b msg ⇝ _ THEN b=true.
```

This approach is extended to exception-handling with the following oracles, which are used in Figure 5.2 (page 5.2).

```
Axiom exn: Type.   Extract Inlined Constant exn ⇒ "exn".
Axiom raise: ∀ {A}, exn → ?? A.  Extract Constant raise ⇒ "raise".
Axiom try_with_any: ∀ {A}, (unit → ?? A) * (exn → ??A) → ??A.
Notation "'TRY' k1 'WITH_ANY' e '⇒' k2" :=
  (try_with_any (fun _ ⇒ k1, fun e ⇒ k2)) ...
```

Here `try_with_any` is implemented in OCaml by

```
let try_with_any (k1, k2) = try k1() with e -> k2 e
```

By parametricity, we can also prove post-conditions on such an exception-handler, provided that these post-conditions are satisfied in all branches of the exception-handler. In other words, {Impure} embeds formal reasoning on exception-handling operators by abstracting these operators as nondeterministic choices. We do not detail this mechanism here, because no example in this document uses such a verified exception-handling operator.

2.3.2 Generic Loops in Coq

This section defines a verified WHILE-loop for partial correctness. Let us first introduce our untrusted oracle for generic loops. We use type `A` as the type of "(potential) reachable states" in the loop (i.e. `A` is the loop invariant). We also use type `B` as the type of "(potential) final states" (i.e. `B` is the post-condition of the loop). Our `loop` oracle is parametrized by an initial state of type `A` and by a function "`step:A -> ??(A+B)`" computing the next state from a non-final state (see the declaration of `loop` in Fig. 2.7). Typically, the Coq type "`(A+B)`" being extracted on OCaml type "`('a, 'b) sum`" defined in Fig. 2.8, we implement this `loop` oracle by the tail-recursive function of Fig. 2.8. Any safe OCaml implementation of a compatible type is also admitted, like the alternative implementation of Fig. 2.9. In this alternative implementation, recursion is not explicit in the code, but is emulated by a reference `fix` containing a function accessing `fix`. Here, the OCaml typechecker is able to verify that this obfuscated piece of code has the expected type.

After defining the `wli` predicate (acronym for "while-loop-invariant"), Fig. 2.7 defines our verified `while` function. It is parametrized by a pure test `cond`, by an impure state-transformer `body`, by a predicate `I` preserved by one iteration of the loop (`wli` condition) and by an initial state `s0`. Parameter `A` (resp. `B`) of `loop` is instantiated on the loop invariant (resp. the postcondition). On this code, the `Program` plugin generates 3 trivial proof obligations:

1. "`I s0 → I s0`".
2. "`(I s0 → I s) → cond s = true → body s⇝s' →(I s0 →I s')`"
   (trivial from `wli` hypothesis).
3. "`(I s0 → I s) → cond s = false → (I s0 → I s) ∧ cond s = false`".

Let us remark that `mk_annot` is necessary to get the appropriate hypothesis on `s'` in the second proof obligations. Fig. 2.10 (page 38) illustrates how to apply this while-loop operator to an iterative computation of Fibonacci numbers.

This technique could be applied to other kind of generic loops. For example, Fig. 5.6 on page 94 defines a generic loop dedicated to refutation of unreachability properties. This generic loop is applied in Fig. 5.7 to check an UNSAT property, as detailed in Section 5.4.3.

2.3.3 Generic (Memoized) Fixpoints in Coq

This section now extends the previous approach to generic fixpoints of functions. The simplest version of such a fixpoint in OCaml is given by `fixp` function in Fig. 2.12. The `fixp` function computes the fixpoint of `step` a function performing one unfolding step of a recursive computation. For example, it is instantiated for the naive recursive computation of Fibonacci numbers as "`fixp (fun fib p -> if p <= 2 then 1 else fib(p-1)+fib(p-2))`".

Of course, with the implementation in Fig. 2.12, this naive computation of Fibonacci numbers performs an exponential number of additions. By using the memoized implementation on Fig. 2.13,

```
Axiom loop: ∀ {A B}, A * (A → ?? (A+B)) → ?? B.

Definition wli{S} (cond: S → bool) (body: S → ??S) (I: S → Prop) :=
  ∀ s, I s → cond s = true → WHEN body s ⇝ s' THEN I s'.

Program Definition
  while {S} cond body (I: S → Prop | wli cond body I) s0
  : ?? {s | (I s0 → I s) ∧ cond s = false}
:= loop (A:={s | I s0 → I s})
        (s0,
            fun s ⇒
            match (cond s) with
            | true ⇒
               DO s' ↜ mk_annot (body s) ;;
               RET (inl (A:={s | I s0 → I s }) s')
            | false ⇒
               RET (inr (B:={s | (I s0 → I s) ∧ cond s = false}) s)
            end).
```

Figure 2.7: Implementation of a WHILE-loop in Coq

```
type ('a, 'b) sum = Coq_inl of 'a | Coq_inr of 'b

let rec loop (a, step) =
  match step a with
  | Coq_inl a' -> loop (a', step)
  | Coq_inr b -> b
```

Figure 2.8: Standard OCaml implementation of oracle `loop` by a tail-recursive loop

```
let loop (a0, step) =
  let fix = ref (fun _ -> failwith "init") in
  (fix := fun a -> match step a with
                   | Coq_inl a' -> (!fix) a'
                   | Coq_inr b -> b);
  (!fix) a0
```

Figure 2.9: Emulating recursion in OCaml with a cyclic higher-order reference

```
(* Specification of Fibonacci numbers by a relation *)
Inductive isfib: Z → Z → Prop :=
 | isfib_base p: p ≤ 2 → isfib p 1
 | isfib_rec p n1 n2:
     isfib p n1 → isfib (p+1) n2 → isfib (p+2) (n1+n2).

(* Internal state of the iterative computation *)
Record iterfib_state := { index: Z; current: Z; old: Z }.

Program Definition iterfib (p:Z): ?? Z :=
  if p ≤? 2
  then RET 1
  else
    DO s ↞
      while (fun s ⇒ s.(index) <? p)
            (fun s ⇒ RET {| index := s.(index)+1;
                            current := s.(old) + s.(current);
                            old:= s.(current) |})
            (fun s ⇒ s.(index) ≤ p
                     ∧ isfib s.(index) s.(current)
                     ∧ isfib (s.(index)-1) s.(old))
            {| index := 3; current := 2; old := 1 |};;
    RET (s.(current)).

(* Correctness of the iterative computation *)
Lemma iterfib_correct p: WHEN iterfib p ⇝ r THEN isfib p r.
```

Figure 2.10: Iterative computation of Fibonacci numbers with the WHILE-loop

```
Parameter beqZ: Z → Z → ?? bool.
Parameter beqZ_correct: ∀ x y, WHEN beq x y ⇝ b THEN b=true → x=y.

Program Definition fib (z: Z): ?? Z :=
  DO f ↞ rec beqZ isfib (fun (fib: Z → ?? Z) p ⇒
    if p ≤? 2
    then RET 1
    else
      let prev := p-1 in
      DO r1 ↞ fib prev ;;
      DO r2 ↞ fib (prev-1) ;;
      RET (r2+r1)) _;;
  (f z).

Lemma fib_correct (x: Z): WHEN fib x ⇝ y THEN isfib x y.
```

Figure 2.11: Computation of Fibonacci numbers with the generic fixpoint

the number of additions remains linear. However, a bug in the implementation of `fixp` like in Fig. 2.14 leads to incorrect results. Here, the implementation in Fig. 2.14 represents an erroneous version of the memoized version of Fig. 2.13 where all recursive results are crashed into a single memory cell (instead of associating each recursive result to its corresponding input into a dedicated memory cell).

```
let fixp (step: ('a -> 'b) -> 'a -> 'b): 'a -> 'b =
  let rec f x = step f x in f
```

Figure 2.12: Standard fixpoint in OCaml

```
let fixp (step: ('a -> 'b) -> 'a -> 'b): 'a -> 'b  =
  let memo = Hashtbl.create 10 in
  let rec f x =
    try Hashtbl.find memo x
    with Not_found -> let r = step f x in (Hashtbl.replace memo x r); r
  in f
```

Figure 2.13: Memoized fixpoint in OCaml

```
let fixp (step: ('a -> 'b ) -> 'a -> 'b): 'a -> 'b =
  let memo = ref None in
  let rec f x =
    match !memo with
    | Some y -> y
    | None -> let r = step f x in (memo:=Some r); r
  in f
```

Figure 2.14: An erroneous memoized fixpoint in OCaml

Hence, Fig. 2.14 gives a *safe* implementation of type `(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b` that does not compute a correct fixpoint, even for partial correctness. This illustrates that the property "*be a correct fixpoint*" cannot be derived by *pure* parametric reasoning (in contrast to the WHILE-loop of Section 2.3.2). However, we build a verified fixpoint operator from any fixpoint oracle of type `(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`, by combining parametricity-by-invariants and (inexpensive) defensive checks. In the case of implementation in Fig. 2.14, the incorrect fixpoint computations will abort (because of the defensive checks). Hence, we declare the following oracle in Coq. And, we build a formally correct fixpoint operator by wrapping this oracle.

```
Axiom fixp: ∀ {A B}, ((A → ?? B) → A → ?? B) → ?? (A → ?? B).
```

Usually, proving the correctness of a (non-tail)recursive functions requires to prove that a given *relation* between inputs and outputs is preserved by the unfolding step of recursion. Here, we need to encode this *binary* relation – called `R` below – into the *unary* invariant `B`. The trick is thus to store both the input (of type `A`) and the output (of type `B`) in this invariant, through the `answ` type below. In the following, `A B: Type` and `R: A → B → Type` are implicit parameters of the formally proved fixpoint operator.

```
Record answ := { input:A; output:B; correct:R input output }.
```

Then, we add a **defensive check** on each recursive result `r` – returned through the oracle – that `(input r)` "*equals to*" the actual input of the call.

Thus, our fixpoint operator is also parametrized by an equality test `beq: A → A → ?? bool` that is expected to satisfy the following formal property.

```
∀ x y, WHEN beq x y ⇝ b THEN b=true → x=y.
```

For example, `beq` could be instantiated by the pointer equality `phys_eq` or a more structural equality test (as detailed later).

Then, we introduce a wrapper `wapply` of the application, such that each recursive call `k` returning a value of type `answ` is converted into a function `(wapply k)` returning a value of type `B`, but with a defensive check that the `input` field equals the `x` parameter.

```
Definition wapply (k: A → ?? answ) (x:A): ?? B :=
  DO a ↜ k x;;
  DO b ↜ beq x (input a);;
  assert_b b msg;;
  RET (output a).
Lemma wapply_correct k x: WHEN wapply k x ⇝ y THEN R x y.
```

The parameter "`step:(A → ?? B) → A → ?? B`", that unfolds one step of recursion, is expected to preserve relation `R`, as formalized by `step_preserv` predicate.

```
Definition step_preserv (step: (A → ?? B) → A → ?? B) := ∀ f x,
  WHEN step f x ⇝ z THEN (∀ x', WHEN f x' ⇝ y THEN R x' y) → R x z.
```

Our proved `rec` operator is thus defined by:

```
Program Definition rec step (H:step_preserv step R): ?? (A → ?? B) :=
  DO f ↜ fixp (B:=answ R)
              (fun k x ⇒
                 DO y ↜ mk_annot (step (wapply k) x);;
                 RET {| input := x; output := 'y |}
              );;
  RET (wapply f).
Lemma rec_correct step (H:step_preserv step R):
  WHEN rec step H ⇝ f THEN  ∀ x, WHEN f x ⇝ y THEN R x y.
```

Fig. 2.11 (page 38) instantiates this `rec` operator on the naive recursive computation of Fibonacci numbers: given any correct `beqZ: Z → Z → ?? bool`, it derives a correct Fibonacci implementation `fib`. Actually, to achieve reasonable performance, `beq` must be chosen at instantiation of operator `rec` according to `fixp` implementation. If `beq` is too much discriminating, then it may reject valid computations. On the contrary, if `beq` inspects too much the structure of its inputs, then it may slow down computations. For example, `phys_eq` is well-suited for the fixpoint implementation in Fig. 2.12. But it is too much discriminating for the fixpoint implementation of Fig. 2.13. Actually, for the latter, `beq` must correspond to the equality test involved in the hash-table implementation: here structural equality. Hence, our approach could be improved by passing `beq` as a parameter of the oracle, which then could use it as the equality test of the hash-table instead of structural equality.

2.4 Make your Oracles as Polymorphic as Possible

Section 2.3 illustrates that we get "free theorems" from higher-order polymorphic oracles. It suggests the following paradigm for designing oracles: "*make your oracles as polymorphic as possible*". This section applies this paradigm on two "toy" examples which introduce the techniques at the heart of

our "realistic" case studies, detailed in next chapters. The example of Section 2.4.1 is reminiscent of the previous example on references (at Fig. 2.4). It illustrates how introducing such an imperative oracle helps in a "concrete" problem like testing *list inclusion*. The example of Section 2.4.2 has a much deeper scope: it shows that our technique of "free theorems" is really helpful on typical co-NP hard problems. Indeed, this second example introduces our *Polymorphic Factory Style* for proving *unsatisfiability of Boolean Conjunctive Normal Forms*.

2.4.1 Testing List Inclusion at Linear Running Time

Let us consider here the following problem. Given a type `A` with an "equality" test, and two lists `l1` and `l2` of type `(list A)`, we would like to check that all elements of `l1` are elements of `l2`. In order to have an efficient implementation, we store all elements of `l2` in a dictionary, and then check that all elements of `l1` are in the dictionary. Given n the maximum size of `l1` and `l2`, we get a running time in $\Theta(n)$ for the amortized average case, if the dictionary is an imperative hash-table – but in $\Theta(n.\log(n))$ if the dictionary is a purely functional binary-search-tree.

Actually, the memoized fixpoint of Section 2.3.3 suggests that – in some cases – we can efficiently embed "for free" an untrusted hash-table into certified code. This is also the case on this example.[14] Hence, on the Coq side, Figure 2.15 declares an oracle for creating a dictionary (by mimicking the style of references at Figure 2.4). Here, `hashcode` is an abstract Coq type, extracted on OCaml `int`.[15] On the OCaml side, we simply wrap the hash-tables provided by the standard library, see Figure 2.16.

Then, we turn this dictionary structure into a "set" structure. Given a type `A` and an "invariant" `inv : A → Prop`, the type `(Sets.t inv)` represents the type of "subsets" of `A`, with elements satisfying `inv`. This type is implemented by dictionaries mapping each element (of type `A`) to a proof that this element satisfies `inv` (see Figure 2.17). Given a list `l2`, function `create` of Figure 2.18 now builds a set which contains only elements in `l2` (by invoking `Sets.empty` and `Sets.add` defined at Figure 2.17). Conversely, function `assert_incl` checks that all elements of `l1` belong to a given set `d` (using `Sets.is_in` of Figure 2.17). At last, function `assert_list_incl` of Figure 2.18 glues these two steps together and provides the expected inclusion test on lists.

In summary, we have embedded a polymorphic untrusted hash-tables of OCaml, in order to get "for free" a kind of certified dictionary. From this certified dictionary, we have derived an efficient and certified inclusion test between lists. Here, note that the formal correctness proof of this efficient imperative dictionary is very lightweight: for example, it is much more lightweight than proofs about binary-search-trees in the Coq standard library. Indeed, it is reduced by parametricity to the polymorphic type of the hash-table.

More generally, in the case studies of this document, we delegate as much computations as possible to external oracles, while reducing invariant preservation proofs to ML polymorphic typechecking. This both reduces the development times and the running times.

2.4.2 A Naive UNSAT Prover on Boolean CNFs

The satisfiability of Boolean CNFs (Propositional Conjunctive Normal Forms) is the archetype of NP-hard problems. Hence, programming and verification techniques adapted for this problem should apply to a large class of problems. Efficiently verifying the "SAT" answer of SAT-solvers is easy: it reduces to evaluate the input CNF within the assignment of Boolean variables found by the SAT-solver

[14] Its full Coq/OCaml code is online at https://github.com/boulme/ImpureDemo

[15] We also embed in Coq the polymorphic `hash` function of OCaml to create hash-codes on the Coq side.

```
Record IDict.hparams (A:Type) := {
  test_eq: A → A → ??bool;
  test_eq_correct: ∀ x y, WHEN test_eq x y ⇝ r THEN r=true → x=y
  hashing: A → ??hashcode;
}.

Record IDict.t (A B:Type) := {
  set: A * B → ?? unit;
  get: A → ?? option B
}.

Axiom IDict.make: ∀ {A B}, IDict.hparams A → ?? IDict.t A B.
```

Figure 2.15: Declaration of Imperative Dictionaries in Coq

```
(* val IDict.make: 'a IDict.hparams -> ('a, 'b) IDict.t *)
let IDict.make (type key) (hp: key IDict.hparams) =
  let module MyHashedType = struct
    type t = key
    let equal = hp.IDict.test_eq
    let hash = hp.IDict.hashing
  end in
  let module MyHashtbl = Hashtbl.Make(MyHashedType) in
  let dict = MyHashtbl.create 10 in
  {
    IDict.set = (fun (k,d) -> MyHashtbl.replace dict k d);
    IDict.get = (fun k -> MyHashtbl.find_opt dict k)
  }
```

Figure 2.16: Implementation of Dictionaries by Hash-Tables in OCaml

(such a verifier is presented in Chapter 5). In contrast, the efficient verification of "UNSAT" answers is challenging: we may still not have a definitive solution to this problem.

This section presents a very lightweight verification of the UNSAT answers of Boolean SAT-solvers. This example is deliberately kept naive: this is our introductory example for the Polymorphic Factory (PFS) design pattern which is developed in further sections. Here, only folklore knowledge on Boolean resolution is required. Chapter 5 presents a more state-of-the-art verifier.

The idea of formally verifying the UNSAT answers of efficient SAT-solvers emerged with the Chaff SAT-solver [ZM03]. Following their idea, we build a verified UNSAT prover with a two-tier architecture: first, an untrusted oracle (i.e. the SAT-solver) that produces a resolution proof or aborts; second, a certified verifier that checks this resolution proof. This is theoretically justified by Theorem 2.1 ensuring that this approach is correct and complete.

Formal specification of the prover in Coq

First, let us recall the definition of CNF (Conjunctive Normal Form).

Definition 2.5 (Conjunctive Normal Form)**.** A *Boolean variable* x is a name and is encoded as a positive integer. A *literal* ℓ is either a variable x or its negation $\neg x$. A *clause* c is a finite disjunction

```
(* Type of "sets" with elements satisfying [inv] *)
Definition Sets.t {A} (inv:A → Prop) := IDict.t A {x | inv x}.

(* building the empty set *)
Definition Sets.empty {A} (hp:IDict.hparams A) {inv:A → Prop}
  : ?? Sets.t inv := IDict.make hp.

(* adding a list of elements [l] -- satisfying [inv] -- to a set [d] *)
Program Fixpoint Sets.add {A} (l:list A) {inv} (d:Sets.t inv)
  : ∀ {H:∀ x, List.In x l → inv x}, ?? unit :=
  match l with
  | nil ⇒ fun H ⇒ RET ()
  | x::l' ⇒ fun H ⇒ d.(IDict.set)(x,x);; Sets.add l' d
  end.

(* testing whether [x] is in a set [d] *)
Definition Sets.is_in{A}(hp:IDict.hparams A)(x:A){inv}(d:Sets.t inv)
  : ?? bool :=
  DO oy ↭ (d.(IDict.get)) x;;
  match oy with
  | Some y ⇒ hp.(test_eq) x (‘y)
  | None ⇒ RET false
  end.

Lemma Sets.is_in_correct A hp (x:A) inv (d:Sets.t inv):
 WHEN Sets.is_in hp x d ⇝ b THEN b=true → inv x.
```

Figure 2.17: Certified “hash-set” operators in Coq

of literals and is encoded as a set of literals. A *CNF* f is a finite conjunction of clauses and is encoded as a list of clauses. A model m of CNF f is a mapping that assigns each variable to a Boolean such that “$[\![f]\!]\ m$” is true – where “$[\![f]\!]\ m$” is the Boolean value obtained by replacing in f each variable x by its value “$m\ x$”. A CNF is said *“SAT”*, if it has a model, and *“UNSAT”* otherwise.

Our Coq definitions of CNF abstract syntax are given in Figure 2.19. These definitions involve external clause identifiers of type `clause_id` without formal semantics. These identifiers are intended to relate clauses to their name in the untrusted oracle. Here, type `clause_id` is opaque for the Coq proof: it remains uninterpreted. In the following, we use the bracket notations [[.]] for both predicates “`sat`” and “`sats`”.

Our goal is to define a `unsatProver` of the following type: given a CNF `f`, if `(unsatProver f)` terminates *normally* then `f` is UNSAT (otherwise `unsatProver` is expected to raise an exception).[16]

```
unsatProver (f: cnf): ?? (∀ m, ¬[[f]] m)
```

A Shallow-Embedded Resolution Checker in Coq

Before introducing our formalization of resolution in Coq, let us recall the following theorem. Actually, in order to certify our prover, we only need to formalize in Coq the correctness proof. The

[16] Alternatively, `unsatProver` could return a Boolean representing “UNSAT” or “FAILED”. Here, as we use an external oracle, the monad is required, and in this case, exceptions simplify programs and proofs and make runtime more efficient.

```
(* returns the set of elements in [l2] *)
Program Definition create {A} (hp: IDict.hparams A) (l2:list A)
  : ?? Sets.t (fun x ⇒ List.In x l2) :=
  DO d ↞ Sets.empty hp (inv:=fun x ⇒ List.In x l2);;
  Sets.add l2 (inv:=fun x ⇒ List.In x l2) d (H:=_);;
  RET d.

(* test inclusion of [l1] into set [d] *)
Fixpoint assert_incl{A}(hp:IDict.hparams A)(l1:list A){inv}(d:Sets.t inv)
  :?? unit :=
  match l1 with
  | nil ⇒ RET ()
  | x::l1' ⇒ DO x_in ↞ Sets.is_in hp x d;;
              assert_b x_in "inclusion fails";;
              assert_incl hp l1' d
  end.
Lemma assert_incl_correct A (hp: IDict.hparams A) l1 inv d:
 WHEN assert_incl hp l1 d ⇝ _ THEN ∀ x, List.In x l1 → inv x.

(* test inclusion of [l1] into set [l2] *)
Definition assert_list_incl {A} (hp: IDict.hparams A) (l1 l2: list A)
  : ?? unit :=
  DO d ↞ create hp l2;;
  assert_incl hp l1 d.
Lemma assert_list_incl_correct A (hp: IDict.hparams A) l1 l2:
 WHEN assert_list_incl hp l1 l2 ⇝ _ THEN List.incl l1 l2.
```

Figure 2.18: Certified and efficient test for list inclusion in Coq

completeness proof only justifies that the design of our prover is expressive enough.

Theorem 2.1 (Refutation correctness & completeness of Forward Resolution)**.** *A CNF f is UNSAT* ***iff*** *the clause $\emptyset$ is derivable by the following derivation rules:*

$$\textsc{Axiom}\ \frac{}{c}\ c \in f \qquad\qquad \textsc{FwdRsl}\ \frac{c_1 \qquad c_2}{(c_1 \backslash \{\ell\}) \cup (c_2 \backslash \{\neg\ell\})}$$

First, we introduce our shallow embedding of resolution proofs in Coq. In our implementation, besides the type `iclause` of the abstract syntax, we have a more computational representation of clauses, called `cclause`, where a clause is represented as two finite sets of positive integers: one for the positive literals, and one for the negative literals. Such finite sets are efficiently defined in the standard library of Coq using radix trees. For the sake of simplicity, the Coq definitions we present here omit this type `cclause` and use `iclause` instead.

Given `f: cnf`, we define the type "`consc⟦`f`⟧`" of clauses that are "*logical consequences*" of f. Actually, type `consc` is parametrized by a set of models `s` and constrains its field `rep` to satisfy all models of s (through `rep_sat` property).

```
Record consc(s: model → Prop): Type :=
  { rep: iclause; rep_sat: ∀ m, s m →⟦snd rep⟧m }.
```

```
Definition var := positive.
Record literal := { is_pos: bool ; ident: var }.
Definition model := var → bool. (* Boolean mapping *)
Definition clause := list literal.  (* syntactic clause *)
Fixpoint sat (c: clause) (m: model): Prop :=
  match c with
  | nil ⇒ False
  | l::c' ⇒ m(ident l)=(is_pos l) ∨ sat c' m
  end.
Definition iclause := clause_id * clause.  (* clause with an id *)
Definition cnf := list iclause.     (* syntactic cnf *)
Fixpoint sats (f: cnf) (m: model): Prop :=
  match f with
  | nil ⇒ True
  | c::f' ⇒ sat (snd c) m ∧ sats f' m
  end.
```

Figure 2.19: Coq definitions of the abstract syntax of a CNF

Then, we define an emptiness test of the following type. Actually, `assertEmpty c` terminates iff `(rep c)` is the empty clause. Otherwise, it raises an exception.

```
assertEmpty {s}: consc s → ??(∀ m, ¬(s m)).
```

Then, we define the forward resolution operator as the following function, called `resol`.

```
resol: ∀{s}, (consc s) → (consc s) → ??(consc s)
```

In its implementation, `(resol c1 c2)` first checks that there exists a *unique* literal `l` such that `l` belongs to `c1` and its negation belongs to `c2`. If this is the case, it applies rule FwdRsl with ℓ :=`l`, c_1 :=`c1` and c_2 :=`c2`. Otherwise it raises an exception, because this is considered as a performance bug of the oracle: any resolvent of `c1` and `c2` by FwdRsl is useless for generating $\emptyset$.

An untrusted solver in Polymorphic LCF Style (PFS)

The `unsatProver` function calls a solver and checks that it has found a valid resolution proof of the input CNF. Actually, it exploits the cooperation mechanism of Coq and OCaml typecheckers in order to make this *untrusted* solver compute directly a logical consequence of the input, through a certified API. This API is called a *Logical Consequence Factory* (LCF) and builds correct-by-construction proofs, without an explicit "proof object" – in the style of the old LCF prover (as discussed in Sect. 1.3.1). The solver is declared in Coq by the `solver` axiom (see below). This function is parametrized by:

- an abstract type of clause: this type – called `C` – is abstract for the untrusted parser but instantiated by "`consc`$[\![f]\!]$" in the Coq proof;
- a logical consequence factory of type "`(resolLCF C)`": this factory allows the oracle to build logical consequences (i.e. new abstract clauses) thanks to `abs_resol` (instantiated by the previous `resol` in the Coq proof).[17]

[17]Note that, type `resolLCF` only appears in input of our oracle: it is permissible as expected. Here, `abs_learn` is declared impure because it may raise exceptions: alternatively, we also could have use an option monad. However, this would probably produce a slightly less efficient extracted code.

- the input CNF f given as a list of "*axioms*", ie abstract clauses of type `C`.

```
Record resolLCF C := { abs_resol: C → C → ?? C;  get_id: C → clause_id }.
Axiom solver: ∀ {C}, (resolLCF C)*list(C) → ?? C.
```

By using the `get_id` function, the solver first builds a map from clause identifiers in the input to their corresponding abstract clause (ie axiom). Then, it maintains this map while creating auxiliary clauses. It is expected to either abort or return an abstract clause witnessing a proof of the empty clause.

Thus, `unsatProver` is simply defined by the code below. It first calls the `mkInput` function that builds the parameters expected by the parser (we omit the details here). Afterwards, `unsatProver` simply invokes the parser and checks that its result is the empty clause. Here, the polymorphism over "abstract clauses" in the OCaml solver ensures that this untrusted code can only forge abstract clauses which are logical consequences of the input.

```
Definition mkInput (f: cnf): resolLCF(consc⟦f⟧) * list(consc⟦f⟧) :=..
Definition unsatProver f: ?? (∀ m, ¬⟦f⟧m) :=
  DO c ↭ solver (mkInput f);; assertEmpty c.
```

This example illustrates that PFS provides a simple, lightweight and efficient API for building a correct-by-construction consequence of the input CNF with an untrusted SAT-solver. However, recent works [Cru+] have surpassed forward resolution with a more efficient proof format. Chapter 5 presents PFS for this more complex proof format.

2.5 Limitations of {Impure} and Future Works

As illustrated by the remainder of this document, the {Impure} library provides an experimental framework to develop case studies combining OCaml untrusted oracles and Coq verified code. However, a tighter integration of OCaml and Coq would be preferable in the future, in order to increase the trustworthiness of such developments. In particular, Definitions 2.3 and 2.4 informally describe two additional typechecking verifications that are required for the soundness of Coq+OCaml verified code.

Below, we detail some issues with the current implementation of {Impure}, and how they could disappear with a more ambitious framework. Section 2.5.1 illustrates that cyclic values of Coq extracted inductive types are not compatible with the pointer equality axiomatized in Coq. Section 2.5.2 explains why it is unsafe to extend Coq+OCaml code in OCaml (instead of programming this extension into Coq thanks to {Impure}). For example, some uncontrolled alias in the OCaml extension may break invariants proved on the Coq side. In particular, the main function of executables should be programmed in Coq instead of OCaml. Last, Section 2.5.3 explains that the current extraction of {Impure} computations still needs to be improved, in order to ensure safe equational reasoning on {Impure} computations, as sometimes used in the {VPL} [#FB14].

2.5.1 The Issues of Cyclic Values

Consider the following Coq code. It defines a type `empty` which is provably empty: the proposition `empty → False` is provable by induction. Thus, any function of `unit→??empty` is proved to never return (normally).

```
Inductive empty: Type:= Absurd: empty → empty.
Lemma never_return_empty (f:unit→??empty): WHEN f() ⇝ _ THEN False.
```

Thus, because `unit→??empty` is permissive, OCaml cyclic values like the `loop` value defined below (with type `empty`) are unsafe (see Definition 2.3).

```
let rec loop = Absurd loop
let f: unit -> empty = fun () -> loop
```

Besides this pathological case, forbidding cyclic values on Coq extracted types is also necessary for the soundness of the physical equality inside Coq introduced at Section 2.2.4. Indeed, otherwise there is an unsoundness issue with axiom `phys_eq_true`.

For example, let us consider the `phys_eq_pred` lemma about type `nat` of Peano's natural number, defined in the standard library. This lemma derives from the fact that `0` is the only `n:nat` such that `pred n = n`.

```
Definition is_zero (n:nat): bool :=
  match n with
  | 0 ⇒ true
  | (S _) ⇒ false
  end.
Lemma phys_eq_pred n:
  WHEN phys_eq (pred n) n ⇝ b THEN b=true → (is_zero n)=true.
```

Let us now consider the following cyclic value – called `fuel` – because some Coq users import such an "infinite fuel" from OCaml in order to circumvent the structural recursion imposed by Coq.

```
let rec fuel: nat = S fuel
```

At runtime, the OCaml test "`pred fuel == fuel`" returns **`true`**, but "`is_zero fuel`" returns **`false`**. This contradicts the `phys_eq_pred` lemma. Hence, in order to formally reason about physical equality in Coq, it is necessary to forbid – in OCaml oracles – cyclic values on types extracted from Coq.

In conclusion, Definition 2.3 forbids oracles to define cyclic values on Coq extracted types. A way to check this property of oracles would consist in adding to the OCaml language an (optional) "`inductive`" tag on OCaml variant types that forbids cyclic values of these types. Then, Coq inductive types would be extracted into OCaml variant types tagged with "`inductive`".

2.5.2 The Issues of Ensuring that Aliases Cannot Break Coq Invariants

This section illustrates interactions between aliases and Coq typing with examples using type `cref` defined in Figure 2.4 page 31 (for the implementation of the oracle given in Figure 2.5). First, we introduce the following Coq code:

```
Definition may_alias{A} (x:cref A) (y:cref nat):?? A:=
  y.(set) 0;; x.(get) ().
```

Now, let us consider `x: cref mydata` where `mydata` is constrained by invariant `bounded`. We are able to prove that `(may_alias x y)` returns a value satisfying this invariant as expressed by `mydata_preserved` lemma below:

```
Record mydata := { value: nat; bounded: value > 10 }.
Lemma mydata_preserved (x: cref mydata) (y: cref nat):
  WHEN may_alias x y ⇝ v THEN v.(value) > 10.
```

Let us remark that `mydata_preserved` property could be *broken* by extending the extracted code with arbitrary OCaml code (even for safe OCaml code). Indeed, in the extracted code, type `mydata` is

extracted to `nat` (because `mydata` is a record type with a single field that is not a proposition). And, given any OCaml "`x:nat cref`", `(may_alias x x)` returns `0` (while changing the contents of `x` for this value).

Actually, Hypothesis 1 states that if safe external OCaml code is "imported" into Coq through a permissive Coq type, such an alias cannot break WLP-theorems proven in Coq. Informally, this hypothesis relies on the typing discipline of Coq itself to forbid any alias that breaks Coq typing: in the Coq code, aliasing references of `(cref mydata)` with references of `(cref nat)` is forbidden. Let me remark here that this does not forbid the presence of all aliases in the Coq code itself. For example, the code below defines a reference `r2` containing a reference `r1`, and runs `(may_alias r2 r1)` which thus changes the contents of the contents of `r2`.

```
Program Definition alias_example (r1: cref nat) : ?? { r | r=r1 } :=
  DO r2 ↜ make_cref (exist (fun r ⇒ r = r1) r1 _);; may_alias r2 r1.
```

Here, through Coq typing, we also formally prove that the result of `(may_alias r2 r1)` is reference `r1`. But, the fact that `r1` contains `0` at the end cannot be formally proven (it depends on `make_cref` implementation).

The preceding example suggests that extending extracted code with an OCaml main function could in theory break some properties proved on the Coq side. More generally (beyond introducing unsafe aliases), writing the main function in OCaml may lead to misusing some functions with dependent types in Coq, because their type after extraction to OCaml cannot prevent such a misuse. It seems thus important to define the main function of executables on the Coq side.

Moreover, the `cref` example illustrates that permissivity checking is a bit more complex than the sketch of Section 2.2.2. In particular, the parameter `A` of type `cref` is both used in input (on `set`) and on output (on `get`). Thus, `(cref nat)` is both permissive and permissible, because type `nat` of Coq coincides with its OCaml extraction (in particular, because of the restriction on cyclic-values, see Section 2.5.1). But `(cref mydata)` are neither permissive nor permissible, because type `mydata` of Coq does not coincide with its extraction.

2.5.3 The Issue of Equality on Impure Computations

When interpreting formal proofs based on the {Impure} library, the user must be aware that only WLP-theorems (defined in Section 2.2.1) have a meaning on the extracted code. In particular, the meaning of Coq equality on impure computations is currently very counterintuitive as explained now.

In the Coq logic, all reduction strategies are equivalent. In particular, for any term `foo`, the Coq logic cannot distinguish between the two following β-convertible terms

`((fun x (_:unit) ⇒ x) foo)` versus `(fun (_:unit) ⇒ foo)`

But in OCaml, the two following expressions are very different

```
((fun x (_:unit) -> x) (print_string "hello"))
versus (fun (_:unit) -> print_string "hello")
```

The first expression prints "hello" whereas the second one is silent. This corresponds to the call-by-value semantics of OCaml.

Let us use this idea to build a counterintuitive Coq theorem. Consider the code in Figure 2.20, that defines the `repeat` operator, a higher-order iterator repeating `n` times a computation `k`. It is applied in `print3` to print a string three times. A careless user could instead provide the wrong `wprint3` implementation, which prints the string only once. Actually, the careful user will have in mind that the parameter `k:??unit` of `wrepeat` is extracted to `k:unit` in OCaml. Thus, at extraction, `k` is `()` –

```
Fixpoint repeat (n:nat) (k: unit → ?? unit): ?? unit :=
  match n with
  | 0 ⇒ RET()
  | S p ⇒ k();; repeat p k
  end.
Definition print3 (s:pstring):?? unit:= repeat 3 (fun _ ⇒ println s).

Fixpoint wrepeat (n:nat) (k: ?? unit): ?? unit :=
  match n with
  | 0 ⇒ RET()
  | S p ⇒ k;; wrepeat p k
  end.
Definition wprint3 (s:pstring): ?? unit := wrepeat 3 (println s).

Lemma wrong_IO_reasoning s: (print3 s)=(wprint3 s).
```

Figure 2.20: Counterintuitive Equality on Impure Computations

the single value of type `unit`. Unfortunately, for the Coq logic, `print3` and `wprint3` are the same as attested by lemma `wrong_IO_reasoning`.

In order to avoid this counterintuitive meaning of equality, we could use an alternative extraction, based on the *deferred* monad below, instead of the identity monad:

$$??A \triangleq \text{unit} \to A \qquad k \rightsquigarrow a \triangleq k() = a \qquad \varepsilon\, a \triangleq \lambda(), a \qquad k_1 \gg\!= k_2 \triangleq \lambda(), k_2\,(k_1())\,()$$

The extraction on the deferred monad is consistent with Coq equality, and more generally with equational reasoning on {Impure} computations, as sometimes used in the {VPL} [#FB14]. But, this extraction induces significant overhead at runtime (and makes the type of OCaml oracles more heavyweight for programming).

A better solution consists in keeping the extraction on the identity monad as much as possible, by building a type system to detect Coq terms that are wrongly extracted in the identity monad (like `wrepeat` above) and extract them with the deferred monad instead. This feature requires a nontrivial type system and a nontrivial modification of the extraction: it is left for future work. Providing equational reasoning over {Impure} computations could be a step toward general-purpose reasoning about side-effects, in the style of InteractionTrees [Xia+20] (see also discussion in Section 2.6).

We conjecture that this counterintuitive equality cannot lead to wrong WLP-theorems, even for the extraction on the identity monad without restriction. In other words, we conjecture that while the results observed at runtime in the deferred monad or in the identity monad can differ, WLP-theorems can only state properties which are satisfied in both extractions.

2.6 Comparison with other Imperative FFIs for Coq

Some alternative FFIs for imperative OCaml code have been mentioned in the previous pages. We now recall these alternative FFIs and point out their differences with {Impure}.

Cybelle[18] is a Coq library, like {Impure}, providing a lightweight extension of the Coq programming language with effects, by combining a monad with extraction to OCaml [Cla+13]. See [Reg19,

[18] https://github.com/clarus/cybele

Chap. 1] for a brief overview. With Cybelle, impure oracles are programmed directly in Coq: they automatically generate certificates (called *prophecies*) allowing to emulate them with pure computations in Coq. This provides a quite generic mechanism to run imperative oracles within Coq tactics, without designing some ad-hoc type of certificates. Note that Coq extraction and OCaml are not in its TCB thanks to the prophecy mechanism. However, Cybelle only embeds a very limited subset of OCaml imperative features: for example, it is not obvious how Cybelle could be extended with physical equality. Moreover, the prophecy mechanism may induce an overhead w.r.t. an ad-hoc certificate format optimized for a particular problem. And, it does not provide any theorem for free on OCaml external code.

FreeSpec[19] is a Coq plugin, based on a variant of free monads [Swi08]—to modularly run and reason about impure computations (like external OCaml functions) inside Coq. In FreeSpec, external OCaml code is trusted: it should respect some contract defined in Coq. FreeSpec allows to fully reason about side-effects and I/O, on the contrary to {Impure}. However, reasoning on impure code seems much more heavyweight than with {Impure}. In particular, currently, it does not provide any theorem for free on OCaml external code.

Similar remarks apply to InteractionTrees[20], a library based on another variant of free monads, which provides co-inductive reasoning about impure computations using simulation relations [Xia+20].

Ideally, an Imperative FFI for Coq would be able to support two kinds of "foreign functions": (1) untrusted external oracles à la {Impure}; (2) trusted external code that corresponds to "primitive observational events", on which we may reason with co-inductive simulations, like within InteractionTrees. However, it seems very difficult to soundly mix both kinds of foreign functions. An hypothetical solution would provide some isolation mechanism able to ensure that untrusted oracles cannot "emit" observational events modeled on the Coq side in an uncontrolled way.

[19] https://github.com/ANSSI-FR/FreeSpec
[20] https://github.com/DeepSpec/InteractionTrees

Chapter 3

FVDP of Instruction Schedulers, by Symbolic Execution with Hash-Consing[†]

While often effective to reduce the overall proof effort, validation a posteriori is not a silver bullet either: many compiler passes are no easier to validate than to prove correct once and for all. Between full compiler verification and full translation validation lies a continuum of combined approaches that remain to be systematically explored.

Xavier Leroy in "*Verified squared: does critical software deserve verified tools?*" [Ler11].

Contents

This chapter presents how the ultra-lightweight FVDP of a hash-consing mechanism, with the {Impure} library, is powerful enough to help the FVDP of an instruction scheduler in CompCert backend. In other words, we study here a two-stages FVDP design. In the first stage, the results of an untrusted instruction scheduler are dynamically verified by symbolic execution. In order to make this dynamic verification efficient, two other untrusted oracles are introduced: this is the second stage of FVDP. One of these oracles is the dictionary factory for list-inclusion, previously detailed in Section 2.4.1. The other one is a generic factory of memoization for hash-consing. This chapter details how this second stage of FVDP is implemented on the top of the {Impure} library (within CompCert). The first stage—implemented using the unsafe FFI of CompCert—is also sketched.

Below, Section 3.1 presents instruction scheduling in CompCert (i.e. the first FVDP stage). Section 3.2 introduces AbstractBasicBlock, the intermediate language in which the core of our scheduling

[†]The technical core of this chapter has been published in [#SBM20].

verifier is defined. At last, Section 3.3 formalizes this scheduling verifier as a simulation test in AbstractBasicBlock, with its hash-consing mechanism (i.e. the second FVDP stage).

3.1 Verified Instruction Scheduling in CompCert

Let me introduce instruction scheduling on a simplified single-issue pipeline with 3 stages, pictured in Figure 3.1: one fetch/decode unit (called DECODE), and two execution units (called EXEC1 and EXEC2). Simple arithmetic instructions such as **ADD** and **SUB** only require one cycle in EXEC1, whereas **LOAD** needs two cycles: one in EXEC1 and one in EXEC2.

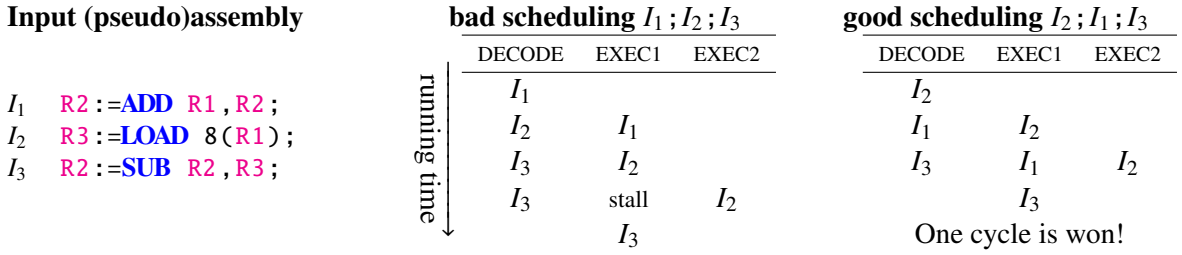


Figure 3.1: Single-Issue Instruction Pipelining with 3 Stages and a **LOAD** of Latency 2.

Figure 3.1 illustrates the execution of a given input assembly program on this pipeline. With the initial order "$I_1\,;I_2\,;I_3$", a stall is introduced by the processor in the pipeline after the decoding of I_3, because its execution in EXEC1 requires register `R3` to be loaded (by instruction I_2). And this will only be the case after having I_2 in EXEC2. Swapping instructions I_1 and I_2 makes this stall disappear (without changing the semantics of the program), because I_1 is executed in EXEC1 while I_2 is in EXEC2. When I_3 enters in EXEC1, `R3` and `R2` are ready. Thus, this second scheduling, by avoiding one stall, makes the program run faster.

Out-of-order processors, such as x86 processors, optimize pipeline usage at runtime with dynamic program scheduling. However, they require a complex control logic, using large CPU die space and energy. In addition, their behavior with respect to execution time may be hard to predict, which is an issue in safety-critical applications where a worst-case execution time (WCET) must be estimated or even justified by a sound analysis [Fra+11]. Last, the complexity of such processors is a source of bugs. [1]

Thus, in-order processors that only implement limited forms of dynamic scheduling (such as parallelizing successive instructions on multiple-issue pipelines) are interesting in the context of embedded systems. And such processors greatly benefit from static scheduling within compilers.

On the previous example, the scheduler [Mic94, Ch. 5] aims to compute $t : \{I_1, I_2, I_3, \$\} \to \mathbb{N}$ assigning a time slot to each instruction, with $t(\$)$ representing the running time (in number of cycles) of the whole sequence. More precisely, the scheduler looks for a t with a minimal $t(\$)$ that satisfies the following constraints, generated from the input program and the target architecture:

Resource Constraints $\quad \forall i \in \mathbb{N},\quad |\{\, x/t(x) = i \,\}| \leq 1 \qquad$ (single-issue pipeline)

Latency Constraints $\quad t(I_3) - t(I_1) \geq 1 \qquad t(I_3) - t(I_2) \geq 2 \qquad t(\$) - t(I_3) \geq 1.$

[1] For instance, Intel's Skylake processor had a bug that crashed programs, under complex conditions [Ler17].

For the above constraints, the optimal assignment ($I_1 \mapsto 1$, $I_2 \mapsto 0$, $I_3 \mapsto 2$, $\$ \mapsto 3$) gives the sequence "$I_2 ; I_1 ; I_3$".

Tristan and Leroy [TL08] have proposed to implement such a scheduler in CompCert by an untrusted oracle, and to dynamically verify the correctness of its answer by **symbolic execution** [Kin76]. For example, it reduces to syntactically check that "$I_1 ; I_2 ; I_3$" (input code) and "$I_2 ; I_1 ; I_3$" (scheduled code) match the same *parallel assignment*—computed by certified symbolic execution:

```
R2:=SUB(ADD R1,R2),(LOAD 8(R1))   ||   R3:=LOAD 8(R1)
```

Indeed, this syntactical check ensures that the scheduled sequence preserves the "local" semantics of the input sequence. However, this does not necessarily ensure that the "global" semantics of the ambiant program is also preserved, if, for example, this ambiant program can enter in the middle of the input sequence through an indirect jump such as "`JMP R2`" (where "`R2`" contains some code address). Hence, such scheduling should only occur inside a notion of *block* with a single (semantic) entry point at its top. Schedules that move instructions across conditional exit points are semantically acceptable, under some various conditions, leading to various scheduling strategies. For example, *super-block scheduling* [Hwu+93] moves some instructions *above* some previous conditional exit points in the block, on the condition that these instructions cannot fail, and they do not modify a register read after having taken those exit points. *Basic-block scheduling* is simpler, by forbidding to move instructions across exit points: scheduling simply occurs inside *basic-blocks*, which are defined as instruction sequences with a single entry point and a single exit point (e.g. a control-flow instruction at the end of the sequence).

Another concern about scheduling is whether it should happen before or after register allocation. After register allocation (i.e. *postpass scheduling*), the scheduling may be limited by some unfortunate register reuse. Before register allocation (i.e. *prepass scheduling*), the instructions to schedule are not yet completely known: in particular, register allocation may induce *register spills*, requiring new load & store instructions. Moreover, the prepass scheduler should take care to not create too many large live ranges, which would increase register pressure, and finally induce more spills. There is also active research in integrating register allocation and instruction scheduling during the same pass, but it does not seem yet clear whether solving these two problems simultaneously can really scale to large generated code [Car+17]. Thus, a standard approach is to implement both kinds of schedulers: a "coarse" prepass scheduler which approximately positions instructions with big latencies; and a "fine" postpass scheduler which precisely optimizes the final program. That is the approach we currently investigate in CompCert.

In their seminal work, Tristan and Leroy [TL08] proposed to extend CompCert with a certified postpass basic-block scheduler, split into (i) an untrusted oracle written in OCaml that computes a scheduling for each *basic block*; (ii) a checker—certified in Coq—that verifies the oracle results by symbolic execution. Unfortunately, their checker has exponential complexity w.r.t. the size of basic blocks, making it slow or even impractical as the number of instructions within a basic block grows. Actually, their approach was neither integrated into CompCert, nor, to my best knowledge, seriously evaluated experimentally, probably due to prohibitive compile times.

In [#SBM20], with Cyril Six and David Monniaux, we have improved Tristan & Leroy's approach on the following points:

- our symbolic execution uses a certified hash-consing of terms: this makes the running times of our scheduling verifier linear w.r.t. the size of basic blocks;
- we certify a scheduler for a VLIW processor: a multiple-issue processor with explicit parallelism in the semantics of the assembly language;
- our scheduler transforms the assembly program, whereas the one of Tristan and Leroy [TL08]

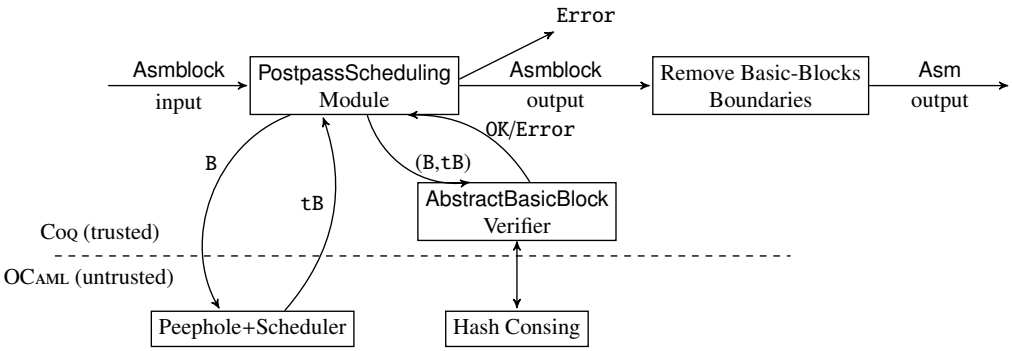


Figure 3.2: Architecture of our Postpass Scheduling on Aarch64

was working at a slightly higher level, not enough precise to produce correct VLIW assembly programs;

- our scheduling checker is able to validate some simple peephole optimizations performed by the scheduler, such as replacing two 64-bit loads by a single 128-bit load;
- the backend of our scheduling checker is defined on a new dedicated intermediate language of CompCert, called AbstractBasicBlock, that is independent of the target processor.

Since this work, Léo Gourdin has ported our postpass scheduling with peepholes to a non-VLIW target: Aarch64 (more precisely, his scheduler targets ARM Cortex-A53, an in-order dual issue pipeline) [Gou21]. Léo's proof directly reuses the AbstractBasicBlock checker (see Figure 3.2). With Cyril Six and David Monniaux, we have also implemented a generic prepass superblock scheduler. Because this scheduler works on superblocks instead of basic blocks, and because it works at a higher level representation than assembly code (called RTL), we have not reused the AbstractBasicBlock checker. This prepass checker still directly reuses the hash-consing factory developed for the AbstractBasicBlock checker [Six21].

Our experimental evaluations shows that adding our scheduling passes does not significantly increase compiling times (the bottleneck of CompCert remains register allocation), and there is a significant speed-up at runtime of the generated code. See [#SBM20].

Previously to our work, symbolic execution with hash-consing has been successfully applied to the *uncertified* translation validation of a wide range of compiler optimizations. For example, see [TGM11]. The FVDP of symbolic execution presented below aims to be a first step toward the FVDP of such powerful translation validators.

3.2 The AbstractBasicBlock IR and its Sequential Semantics

The remainder of this chapter details AbstractBasicBlock and its scheduling checker. AbstractBasicBlock is an IR (Intermediate Representation) dedicated to verification of the results of scheduling oracles operating on basic blocks. It also helps to check bundling oracles (for VLIW targets), but we skip this part here: see [#SBM20] for details. This IR is only used for verification: there are only translation from assembly to AbstractBasicBlock (see Figure 3.2), but no translation from AbstractBasicBlock to another IR of CompCert. AbstractBasicBlock is independent of the target processor and from the remainder of CompCert. Because of this good feature, the following description of AbstractBasicBlock intends to be self-contained, and does not require understanding other parts of CompCert. (Presenting in details the simulation proof of the whole input program by the whole scheduled program from the correctness of AbstractBasicBlock scheduling checker, is beyond the scope of

this chapter: see [#SBM20].)

Section 3.2.1 illustrates how assembly instructions are compiled into AbstractBasicBlock: this introduces the syntax of AbstractBasicBlock instructions. Section 3.2.2 formally defines this syntax and its associated semantics. Section 3.3 presents the *simulation test*, which checks that the sequential semantics of basic blocks is preserved by scheduling.

3.2.1 Introduction through the Translation from Assembly

AbstractBasicBlock defines a (deeply embedded) language for representing the semantics of single assembly instructions as the assignment of one or more pseudoregisters. For example, an instruction "add r_1, r_2, r_3" is represented as an assignment "r_1 := add$[r_2, r_3]$". Hence, AbstractBasicBlock distinguishes syntactically which pseudoregisters are in input or output of each instruction. Moreover, it gives to all operations (including load/store and control-flow ones) a single signature "`list exp` → `exp`". A binary operation such as `add` will just dynamically fail, if applied to an unexpected list of arguments. This makes the syntax of AbstractBasicBlock very simple.

Let us consider less straightforward examples. Our translation to AbstractBasicBlock represents the whole memory as a single pseudoregister called here m. Hence, instruction "`load` r_1, r_2, i" (where i is an integer constant representing offset) is encoded an assignment "r_1 := (`load` i)$[m, r_2]$" where the underlying operation is "(`load` i)". In other words, the syntax of AbstractBasicBlock provides an infinite number of operations "(`load` i)" (one for each i). Similarly, a "`store` r_1, r_2, i" is encoded an assignment "m := (`store` i)$[m, r_1, r_2]$" reflecting that the whole memory is potentially modified.

We also encode control-flow instructions in AbstractBasicBlock: a control-flow instruction modifies the special register `PC` (the program counter). Actually, in CompCert assembly languages, we consider that each basic block ends with a control-flow instruction: when the latter is implicit in the assembly code, it corresponds to the increment of `PC` by the size of the basic block. Hence, in the translation of instructions to AbstractBasicBlock, each control-flow instruction performs at least the assignment "`PC` := (`incr` i)[`PC`]" where i is an integer representing the size of the basic block. Typically, a conditional branch such as "`lt` r, l" (where l is the label and r a register) is translated as the *sequence* of two assignments in AbstractBasicBlock:

$$\texttt{PC} := (\texttt{incr}\ i)[\texttt{PC}]\ ;\quad \texttt{PC} := (\texttt{lt}\ l)[\texttt{PC}, r]$$

It could equivalently be coded as the assignment "`PC` := (`lt` l)[(`incr` i)[`PC`], r]". However, it could seem more convenient to insert the incrementation of `PC` before the assignments specific to each control-flow instruction. A more complex control-flow instruction such as "`call` f" (where f is a function symbol) modifies two registers: `PC` and `RA` (the return address). Hence "`call`f" is translated as the *sequence* of 3 assignments in AbstractBasicBlock:

$$\texttt{PC} := (\texttt{incr}\ i)[\texttt{PC}]\ ;\quad \texttt{RA} := \texttt{PC}\ ;\quad \texttt{PC} := (\texttt{cte}\ \text{address}_f)[]$$

Finally, CompCert assembly languages contain instructions modifying several pseudoregisters in parallel. One of them is an atomic parallel load from a 128-bit memory word into two contiguous (and adequately aligned) destination registers d_0 and d_1. These two destination registers are distinct from each other by construction—but not necessarily from the base address register a. These parallel assignments are expressed in the sequential semantics of AbstractBasicBlock *instructions* with the special `Old` operator of AbstractBasicBlock *expressions*: an expression "(`Old` e)" evaluates "e" in the initial state of the surrounding AbstractBasicBlock instruction.[2] Hence, the parallel load of 128-bit

[2]Such an operator `Old` is quite standard in Hoare logic assertions. For example, see the ACSL annotation language of Frama-C [Kir+15].

words is given in terms of two loads of 64-bit words:[3]

$$d_0 \coloneqq (\texttt{load}\ i)[m, a]\ ;\quad d_1 \coloneqq (\texttt{load}\ (i+8))[m, (\texttt{Old}\ a)]$$

Similarly, CompCert assembly languages provide a pseudoinstruction `freeframe` modifying both the memory and some registers. It is used in the epilogue of functions. In the semantics, `freeframe` modifies the memory m by deallocating the current stack frame in the memory model of CompCert. It also updates register `SP` (the stack pointer) accordingly and destroys the contents of a scratch register called here *tmp*. The modifications to `SP` and m are performed in "parallel", since `SP` indicates the current stack frame in m, and the new value of `SP` is read from this stack frame. For the pseudoinstruction "`freeframe` i_1 i_2" (where i_1 and i_2 are two integers), our translation to AbstractBasicBlock introduces two intermediate operations: first, "(`freeframe_m` i_1 i_2)" for the effect on memory, and second, "(`freeframe_SP` i_1 i_2)" for the effect on the stack pointer. Then, the pseudoinstruction "`freeframe` i_1 i_2" is translated as the *sequence* of 3 assignments in AbstractBasicBlock:

$$\begin{aligned} m &\coloneqq (\texttt{freeframe\_m}\ i_1\ i_2)[\texttt{SP}, m]\ ; \\ \texttt{SP} &\coloneqq (\texttt{freeframe\_SP}\ i_1\ i_2)[\texttt{SP}, (\texttt{Old}\ m)]\ ; \\ tmp &\coloneqq \texttt{Vundef}[] \end{aligned}$$

In conclusion, each instruction of CompCert assembly languages is translated into a sequence of assignments, where some of these assignments modify several pseudoregisters in "parallel" thanks to the special `Old` operator. We speak about *atomic sequences of assignments* (ASA): these sequences represent atomic instructions which can themselves be combined either sequentially or in parallel (for verification of VLIW bundles). An *abstract basic block* is a sequence of ASA.

The compilation of assembly basic blocks toward AbstractBasicBlock must produce a *bisimulable* basic block: namely, the compilation must both preserve well-defined and undefined behaviors. This is necessary for the proof of the scheduling correctness. Indeed, this proof consists in building the commutative diagram at the right-hand side of Figure 3.3, in order to deduce the simulation at the assembly level from the one at the AbstractBasicBlock level, because on the top rectangle of the diagram: the downward way of the bisimulation gives the path from B_1, whereas the upward way gives the path to B_2. The remainder of this chapter details how is built the bottom rectangle of the diagram. See [#SBM20] for more details on the top rectangle.

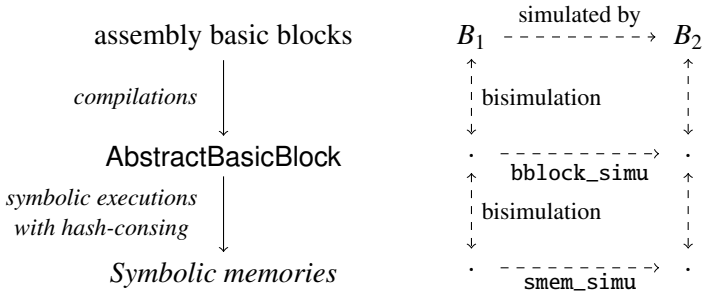


Figure 3.3: Proving that the input block B_1 is simulated by the scheduled block B_2

[3] A benefit of this translation is that our scheduling oracle may replace two loads of 64-bit words into one load of a 128-bit words, and our verifier is able to check "for free" whether the replacement is semantically correct.

3.2.2 Syntax and Sequential Semantics

```
(* Syntax parametrized by type R.t of registers and op of operators *)
Inductive exp := PReg(x:R.t) | Op (o:op) (le:list_exp) | Old (e:exp)
     with list_exp :=...
Definition inst := list (R.t * exp). (* inst = ASA *)
Definition bblock := list inst.

(* Semantic parameters and auxiliary definitions *)
Parameter value genv: Type.
Parameter op_eval: genv → op → list value → option value.
Definition mem := R.t → value. (* concrete memories *)
Definition assign (m:mem) (x:R.t) (v:value): mem :=
  fun y ⇒ if R.eq_dec x y then v else m y.

(* Sequential Semantics *)
Fixpoint exp_eval (ge: genv) (e: exp) (m old: mem): option value :=
 match e with
 | PReg x ⇒ Some (m x)
 | Old e ⇒ exp_eval ge e old old
 | Op o le ⇒ SOME lv ← list_exp_eval ge le m old IN op_eval ge o lv
 end
with list_exp_eval ge (le: list_exp) (m old: mem): option (list value) :=
  ...
Fixpoint inst_run (ge: genv) (i: inst) (m old: mem): option mem :=
 match i with
 | nil ⇒ Some m
 | (x,e)::i' ⇒ SOME v' ← exp_eval ge e m old IN
                inst_run ge i' (assign m x v') old
 end.
Fixpoint run (ge: genv) (p: bblock) (m: mem): option mem :=
 match p with
 | nil ⇒ Some m
 | i::p' ⇒ SOME m' ← inst_run ge i m m IN run ge p' m'
 end.
```

Figure 3.4: Syntax and Sequential Semantics of AbstractBasicBlock

We explain below the formal definition of AbstractBasicBlock syntax and its sequential semantics, formalized in Figure 3.4.[4] Its syntax is parametrized by a type `R.t` of pseudoregisters (positive integers in practice) and a type `op` of operators. Its semantics is parametrized by a type `value` of values, a type `genv` for global environments, and a function `op_eval` evaluating operators to an "`option value`".

Let us introduce the semantics from function `run`, its entry point. Function `run` defines the semantics of a `bblock` by sequentially iterating over the execution of instructions, called `inst_run`. The `inst_run` function takes two memory states as input: `m` as the current memory, and `old` as the initial state of the instruction run (the duplication is carried out in `run`). It invokes the evaluation of an expression, called `exp_eval`. Similarly, the `exp_eval` function takes two memory states as input:

[4] From here, "`SOME` $v \leftarrow e_1$ `IN` e_2" means "**`match`** e_1 **`with`** `Some` $v \Rightarrow e_2$ | `_` ⇒ `None` **`end`**"

the current memory is replaced by `old` when entering under the `Old` operator.

3.3 A Generic Simulation Test for FVDP of Instruction Schedulers

The sequential simulation of a block `p1` by a block `p2` is defined by the `bblock_simu` pre-order, pictured in Figure 3.3. This pre-order is formally defined by:

```
Definition bblock_simu (p1 p2: bblock): Prop :=
  ∀ ge m, (run ge p1 m) <> None → (run ge p1 m) = (run ge p2 m).
```

We have implemented the following simulation test: it takes two blocks `p1` and `p2`, and returns a Boolean, such that if this latter is true then `(bblock_simu p1 p2)`. This test is largely inspired by the list-scheduling verifier of Tristan and Leroy [TL08], but with two major differences. First, they define their verifier for the Mach IR, while ours defined for AbstractBasicBlock is slightly more generic. Second, we use hash-consing in order to avoid a combinatorial explosion of the test.

As in [TL08], the simulation test symbolically executes each AbstractBasicBlock code and compares the resulting *symbolic memories* (Fig. 3.3). A symbolic memory roughly corresponds to a parallel assignment equivalent to the input block. More precisely, this symbolic execution computes a term for each pseudoregister assigned by the block: this term represents the final value of the pseudoregister as a function of its initial value. As in Tristan and Leroy [TL08], our simulation test symbolically executes each block, and then simply compares the resulting *symbolic memories*.

Example 3.1 (Equivalence of symbolic memories)**.** Let us consider the two blocks B_1 and B_2 below:
(B_1) $r_1 := r_1 + r_2;\ r_3 := \texttt{load}[m, r_2];\ r_1 := r_1 + r_3$
(B_2) $r_3 := \texttt{load}[m, r_2];\ r_1 := r_1 + r_2;\ r_1 := r_1 + r_3$
They are both equivalent to this parallel assignment $r_1 := (r_1 + r_2) + \texttt{load}[m, r_2] \parallel r_3 := \texttt{load}[m, r_2]$.
Indeed, B_1 and B_2 bisimulate (they simulate each other).

Collecting only the final term associated with each pseudoregister is actually incorrect: an incorrect scheduling oracle could insert additional failures. The symbolic memory must thus also collect a list of all intermediate terms on which the sequential execution may fail and that have disappeared from the final parallel assignment. See Example 3.2 below. Formally, the symbolic memory and the input block must be bisimulable as pictured on Figure 3.3.

Example 3.2 (Simulation on symbolic memories)**.** Consider:
(B_1) $r_1 := r_1 + r_2;\ r_3 := \texttt{load}[m, r_2];\ r_3 := r_1;\ r_1 := r_1 + r_3$
(B_2) $r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$
Both B_1 and B_2 lead to the same parallel assignment $r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$.

However, B_1 is simulated by B_2 whereas the converse is not true. This is because the memory access in B_1 may cause its execution to fail, whereas this failure cannot occur in B_2. Thus, the symbolic memory of B_1 should contain the term "$\texttt{load}[m, r_2]$" as a potential failure. We say that *a symbolic memory d_1 is simulated by a symbolic memory d_2* if and only if their parallel assignment are equivalent, and the list of potential failures of d_2 is included in the list of potential failures of d_1.

Our formal development is decomposed into two parts using a *data-refinement* style. In a first part, presented in Section 3.3.1, we define a model of the symbolic execution and the simulation test. In a second part, sketched by Section 3.3.3, we refine this model with efficient data-structures and algorithms, involving hash-consing of terms. Indeed, as illustrated by the previous examples, without a mechanism dealing efficiently with duplication of terms, symbolic execution produces terms that

may be exponentially big w.r.t to the size of the source block. Our technique for hash-consing terms is explained in Section 3.3.2.

3.3.1 A Model of our Simulation Test

The principle of *symbolic execution* was first introduced by King [Kin76]. "Symbolic execution" refers to *how to* compute "symbolic memories" (and not to *what* they are) : mimicking the concrete execution while replacing operations on "concrete memories" by operations on "symbolic memories".

In this analogy, "values" are replaced by "*symbolic values*", which are actually terms evaluated in the *initial memory*. Hence, our type `term` of terms—defined below—is similar to type `exp` without the `Old` operator: in a term, a pseudoregister represents its value in the initial memory of block execution.

```
Inductive term := Input (x:R.t) | App (o: op) (l: list_term)
with list_term :=...

Fixpoint term_eval (ge: genv) (t: term) (m: mem): option value :=...
```

In our model, the symbolic execution of a block is a function `bblock_smem: bblock → smem`, where a symbolic memory of type `smem` is the pair of a predicate `pre` expressing at which condition the intermediate computations of the block do not fail, and of a parallel assignment `post` on the pseudoregisters.

```
Record smem:= {pre: genv → mem → Prop; post: R.t → term}.
```

Then, the bisimulation property between the symbolic memory and sequential execution is expressed by the `bblock_smem_correct` lemma below. It uses the `smem_correct` predicate, relating the symbolic memory `d` with an initial memory `m` and a final optional memory `om`.

```
Definition smem_correct ge (d: smem) (m: mem) (om: option mem): Prop :=
  ∀ m', om=Some m'
        ↔ (d.(pre) ge m ∧ ∀ x, term_eval ge (d.(post) x) m = Some (m' x)).
Lemma bblock_smem_correct ge p m:
  smem_correct ge (bblock_smem p) m (run ge p m).
```

By using this lemma, we transfer the notion of simulation of block executions into the simulation of symbolic memories, through the predicate `smem_simu` defined in Figure 3.6. In particular, proposition `(smem_valid ge d m)` holds iff the underlying execution does not return a `None` result from the initial memory `m`. Theorem `bblock_smem_simu` in Figure 3.6 thus formalizes the bottom rectangle of Figure 3.3 diagram in the abstract model of symbolic execution.

Internally, as coined in the name of "symbolic execution", `bblock_smem` mimics `run`, by replacing operations on memories of type `mem` by operations of type `smem`: these operations on the symbolic memory are given in Fig. 3.5. The initial symbolic memory is defined by `smem_empty`. The evaluation of expressions on symbolic memories is defined by `exp_term`: it outputs a term (i.e. a *symbolic value*). Also, the assignment on symbolic memories is defined by `smem_set`. To conclude, starting from `smem_empty`, the symbolic execution preserves the `smem_correct` relation w.r.t the initial memory and the current (optional) memory, on each assignment.

3.3.2 Formally Verified Hash-Consed Terms in Coq

Hash-consing is a standard technique of imperative programming, which in our case has two benefits: it avoids duplication of structurally equal terms in memory, and importantly, it reduces (expansive)

```
(* initial symbolic memory *)
Definition smem_empty :=
  {| pre:=(fun _ _ ⇒ True); post:=(fun x ⇒ Input x) |}.

(* symbolic evaluation of the right-hand side of an assignment *)
Fixpoint exp_term (e: exp) (d old: smem) : term :=
  match e with
  | PReg x ⇒ d.(post) x
  | Op o le ⇒ App o (list_exp_term le d old)
  | Old e ⇒ exp_term e old old
  end
with list_exp_term (le: list_exp) (d old: smem) : list_term :=...

(* effect of an assignment on the symbolic memory *)
Definition smem_set (d:smem) x (t:term) :=
  {| pre:=(fun ge m ⇒ term_eval ge (d.(post) x) m <> None
           ∧ (d.(pre) ge m));
     post:=(fun y ⇒ if R.eq_dec x y then t else d.(post) y) |}.
```

Figure 3.5: Basic Operations of the Symbolic Execution in the Abstract Model

```
Definition smem_valid ge (d: smem) (m:mem): Prop :=
  d.(pre) ge m ∧ ∀ x, term_eval ge (d.(post) x) m <> None.

Definition smem_simu (d1 d2: smem): Prop :=
   (∀ ge m, smem_valid ge d1 m → smem_valid ge d2 m)
 ∧ (∀ ge m x, smem_valid ge d1 m →
          term_eval ge (d1.(post) x) m = term_eval ge (d2.(post) x) m).

Theorem bblock_smem_simu p1 p2:
   smem_simu (bblock_smem p1) (bblock_smem p2) → bblock_simu p1 p2.
```

Figure 3.6: Proving the Bottom Rectangle of Figure 3.3 Diagram (in the Abstract Model).

structural equality tests over terms, to (very cheap) *pointer equality* tests. In our verified backend, we thus import pointer equality from OCaml from the {Impure} library, as axiomatized in Section 2.2.4.

A Generic and Verified Factory of Memoizing Functions for Hash-Consing Hash-consing is a fundamentally impure construction, and it is not easy to retrofit it into a pure language. Braibant, Jourdan, and Monniaux [BJM14] propose several approaches for hash-consing in Coq and in code extracted from Coq to OCaml. However, we need weaker properties than what they aim for. They wish to use physical equality (or equality on an "identifier" type) as equivalent to semantic equality; they use this to provide a fast equality test for Binary Decision Diagrams (BDD)—two Boolean functions represented by reduced ordered binary decision diagrams are equal if and only if the roots of the diagrams are physically the same. In contrast, we just need physical equality to imply semantic equality. This allows for a lighter approach.

Hash-consing consists in memoizing the constructors of some inductive data-type —such as the terms described above—in order to ensure that two structurally equal terms are allocated to the same object in memory. In practice, this technique simply replaces the usual constructors of the data-type

```
(* Parameters for hash-consed types *)
Record hashP (A:Type) := {
  hash_eq: A → A → ?? bool;
  hashing: A → ?? hashcode;
  set_hid: A → hashcode → A
}.

(* Interface of the untrusted OCaml oracle *)
Axiom xhCons: ∀ {A}, hashP A → ??(A → ??A).

(* Defensive wrapper of the untrusted oracle *)
Definition hCons {A} (hp: hashP A): ??(A → ??A)  :=
  DO hC ↞ xhCons hp;;
  RET (fun x ⇒
       DO y ↞ hC x;;
       DO b ↞ hp.(hash_eq) x y;;
       (* Below: exception raised if Boolean [b] is [false] *)
       assert_b b "xhCons: hash-eq differs";;
       RET y).

(* Correctness property of our verified hash-consing *)
Lemma hCons_correct A (hp: hashP A):
  WHEN hCons hp ⇝ hC THEN ∀ (R: A → A → Prop),
  (∀ x y, WHEN hp.(hash_eq) x y ⇝ b THEN b=true → R x y)
  → ∀ x, WHEN hC x ⇝ y THEN R x y.
```

Figure 3.7: Formally Verified Hash-Consing Factory

by *smart constructors* that perform memoization. Memoization is usually delegated to a dedicated function in turn generated from a generic factory.

On the top of the {Impure} library, we have defined in Coq a generic and verified memoization factory. This factory is inspired by that of Filliâtre and Conchon [FC06] in OCaml. However, whereas their factory was not formally verified, ours satisfies a simple correctness property that is formally verified in Coq (and shown sufficient for the formal correctness of our simulation test). Actually, we use an external *untrusted* OCaml oracle that creates memoizing functions and we only *dynamically* check that these untrusted functions behave *observationally* like an identity. Let us insist on this point: the formal correctness of our memoization factory does not assume nor prove that our oracle is correct; it only assumes that the embedding of OCaml trusted pointer equality is correct (see Section 2.2.4). We now detail a slightly simplified version of this factory.[5] Its Coq code is provided in Figure 3.7.

Our generic memoization factory is parametrized by a record of type `(hashP A)`, where `A` is the type of objects to memoize. In Fig. 3.7, `hashcode` is an abstract data type on the Coq side, extracted as an OCaml `int`. Function `hash_eq` is typically a fast equality test, for comparing a new object to already memoized ones in smart constructors. This test typically compares the children of the root node w.r.t pointer equality (the example for terms is given below by `term_hash_eq` function). Function `hashing` is expected to provide a unique hashcode for data that are equal modulo `hash_eq`. Finally, `set_hid` is invoked by memoizing functions to allocate a fresh and unique hash-tag to new

[5] Our actual factory also provides some debugging features, which are useful for printing a trace when the whole simulation test fails. We omit these implementation details in this presentation.

objects (this hash-tag is used by efficient implementations of `hashing`).

The details on `hashing` and `set_hid` are only relevant for efficiency: these functions are simply ignored in our formal proofs. Hence, given such `(hashP A)` structure, our OCaml oracle `xhCons` returns a (fresh) memoizing function of type `(A → ??A)`.

Such a memoizing function of type `(A → ??A)` is expected to behave as an identity w.r.t `hash_eq`. Actually, as we do not trust `xhCons`, we dynamically check this property.[6] Hence, our *verified* generic memoization factory in Coq –called `hCons`—simply wraps each function returned by `xhCons` with this defensive check: it raises an exception if the memoizing function does not return a result equal to its input (w.r.t `hash_eq`). We recall that the notation "`DO` $x \leftsquigarrow e_1$ `;;` e_2" in `hCons` stands for a *bind* operation of the may-return monad of the {Impure} library. Moreover, "`RET` e" is the *unit* of this monad. Function "`assert_b`" is also provided by {Impure}.

Finally, we are able to formally prove the (trivial) correctness property `hCons_correct`, which is sufficient in our development to reason about hash-consing. Here, the relation `R` is typically an equivalence under which we want to observe hash-consed objects.

Smart Constructors for Hash-Consed Terms In our development, we need hash-consing on two types of objects: `term` and `list_term`, because they are mutually inductive. First, we redefine type `term` and `list_term` into `hterm` and `list_hterm` by inserting a hash-tag—called below `hid`—at each node.

```
Inductive hterm := | Input (x:R.t) (hid:hashcode)
                   | App (o: op) (l: list_hterm) (hid:hashcode)
with list_hterm := | LTnil (hid:hashcode)
                   | LTcons (t:hterm) (l:list_hterm) (hid:hashcode).
```

Thus, we also have to redefine `term_eval` and `list_term_eval` for their "`hterm`" versions. Note that these functions simply ignore hash-tags.

```
Fixpoint hterm_eval (ge: genv) (t: hterm) (m: mem): option value :=
  match t with
  | Input x _ ⇒ Some (m x)
  | App o l _ ⇒ SOME v ← list_hterm_eval ge l m IN op_eval ge o v
  end
with list_hterm_eval ge (l: list_hterm) (m: mem): option (list value) :=...
```

Then, we define two records of type `(hashP hterm)` and `(hashP list_hterm)`. Below, we only detail the case of `(hashP hterm)`, as the `(hashP list_hterm)` case is similar. First, the `hash_eq` field of `(hashP hterm)` is defined as function `term_hash_eq` below. On the `Input` case, we use structural equality over pseudoregisters. On the `App` case, we use an equality `op_eq` on type `op` in parameters of the simulation test, and we use pointer equality over the list of terms.

```
Definition term_hash_eq (ta tb: hterm): ?? bool :=
  match ta, tb with
  | Input xa _, Input xb _ ⇒ RET (if R.eq_dec xa xb then true else false)
  | App oa lta _, App ob ltb _ ⇒
     DO b ⇜ op_eq oa ob ;;
     if b then phys_eq lta ltb else RET false
  | _,_ ⇒ RET false
  end.
```

[6] As `hash_eq` is expected to be constant-time, this dynamic check only induces a small overhead.

Second, the `hashing` field of `(hashP hterm)` is defined as function `term_hashing` below. This function uses an untrusted oracle "`hash: ∀{A}, A → ??hashcode`" extracted as the polymorphic `Hashtbl.hash` of the OCaml standard library. It also uses `list_term_get_hid` defined below—that returns the hash-tag at the root node. To ensure memoization efficiency, two terms that are distinct w.r.t `term_hash_eq` are expected to have distinct `term_hashing` with a high probability.[7] This property relies here on the fact that when `term_hashing` is invoked on a node of the form "`(App o l _)`", the list of terms `l` is already memoized, and thus `l` is the unique `list_hterm` associated with the hash-tag `(list_term_get_hid l)`.

```
Definition list_term_get_hid (l: list_hterm): hashcode :=
  match l with
  | LTnil hid ⇒ hid
  | LTcons _ _ hid ⇒ hid
  end.

Definition term_hashing (t:hterm): ?? hashcode :=
  match t with
  | Input x _ ⇒ DO hc ↞ hash 1;; DO hv ↞ hash x;;
                hash [hc;hv]
  | App o l _ ⇒ DO hc ↞ hash 2;; DO hv ↞ hash o;;
                hash [hc;hv;list_term_get_hid l].
  end.
```

Finally, the `set_hid` field of `(hashP hterm)` updates the hash-tag at the root node. It is defined by:

```
Definition term_set_hid (t: hterm) (hid: hashcode): hterm :=
  match t with
  | Input x _ ⇒ Input x hid
  | App op l _ ⇒ App op l hid
  end.
```

Having defined two records of type `(hashP hterm)` and `(hashP list_hterm)` as sketched above, we can now instantiate `hCons` on each of these records. We get two memoizing functions `hC_term` and `hC_list_term` (Fig. 3.8). The correctness property associated with each of these functions is derived from `hCons_correct` with an appropriate relation `R`: the semantic equivalence of terms (or list of terms). These memoizing functions and their correctness properties are parameters of the code building `hterm` and `list_hterm` described below.

```
Variable hC_term: hterm → ?? hterm.
Hypothesis hC_term_correct: ∀ t, WHEN hC_term t ⇝ t' THEN
  ∀ ge m, hterm_eval ge t m = hterm_eval ge t' m.

Variable hC_list_term: list_hterm → ?? list_hterm.
Hypothesis hC_list_term_correct: ∀ lt, WHEN hC_list_term lt ⇝ lt' THEN
  ∀ ge m, list_hterm_eval ge lt m = list_hterm_eval ge lt' m.
```

Figure 3.8: Memoizing Functions for Hash-Consing of Terms (and List of Terms)

Indeed, these functions are involved in the smart constructors of `hterm` and `list_hterm`. Below,

[7] Two terms equal w.r.t `term_hash_eq` *must* also have the same `term_hashing`.

we give the smart constructor—called `hApp`—for the `App` case with its correctness property. It uses a special hash-tag called `unknown_hid` (never allocated by our `xhCons` oracle). The three other smart constructors are similar.

```
Definition hApp (o:op) (l: list_hterm) : ?? hterm :=
  hC_term (App o l unknown_hid).
Lemma hApp_correct o l: WHEN hApp o l ⇝ t THEN ∀ ge m,
  hterm_eval ge t m = (SOME v ← list_hterm_eval ge l m IN op_eval ge o v).
```

In the next section, we only build `hterm` and `list_hterm` by using the smart constructors defined above. This ensures that we can replace the structural equality over type `hterm` by the physical equality. However, this property does not need to be formally proved (and we have no such formal proof, since this property relies on the correctness of our untrusted memoization factory).

3.3.3 Implementing the Simulation Test

Our implementation can be decomposed in two parts. First, we implement the symbolic execution function as a *data-refinement* of the `bblock_smem` function of Section 3.3.1. Then, we exploit the `bblock_smem_simu` theorem to derive the simulation test. In other words, the bottom rectangle of Figure 3.3 diagram is transferred from the abstract model to the implementation of symbolic execution by data-refinement: this leads to theorem `bblock_simu_test_correct` in Fig. 3.11.

Refining Symbolic Execution with Hash-Consed Terms Our symbolic execution builds hash-consed terms. It invokes the smart constructors of Section 3.3.2, and is thus itself parametrized by the memoizing functions `hC_term` and `hC_list_term` defined in Figure 3.8. Note that our simulation test will ultimately perform two symbolic executions, one for each block. Furthermore, these two symbolic executions share the same memoizing functions, leading to an efficient comparison of the symbolic memories through pointer equality. In the following paragraph, functions `hC_term` and `hC_list_term` remain implicit parameters as authorized by the section mechanism of Coq.

Figure 3.9 refines the type `smem` of symbolic memories into a type `hsmem`. The latter involves a dictionary with pseudoregisters of type `R.t` as keys, and terms of `hterm` as associated data. These dictionaries of type `(PDict.t hterm)` are implemented as prefix-trees, through the `PositiveMap` module of the Coq standard library.

Figure 3.9 also relates type `hsmem` to type `smem` (in a given environment `ge`), by a relation called `smem_model`. The `hpre` field of the symbolic memory is expected to contain a list of all the potential failing terms in the underlying execution. Hence, predicate `hsmem_valid` gives a precondition on the initial memory `m` ensuring that the underlying execution will not fail. This predicate is thus expected to be equivalent to the `smem_valid` predicate of the abstract model. Function `hsmem_post_eval` gives the final (optional) value associated with pseudoregister `x` from the initial memory `m`: if `x` is not in the `hpost` dictionary, then its associated value is that of the initial memory (it is expected to be unassigned by the underlying execution). This function is thus expected to simulate the evaluation of the symbolic memory of the abstract model.

Hence, `smem_model` is the (data-refinement) relation for which our implementation of the symbolic execution simulates the abstract model of Section 3.3.1. Figure 3.10 provides an implementation of the operations of Figure 3.5 that preserves the data-refinement relation. The smart constructors building hash-consed terms are invoked by the `exp_hterm` (i.e., the evaluation of expressions on symbolic memories). The `hsmem_set` implementation (Fig. 3.10) is an intermediate refinement toward the actual implementation, improving on two points. First, in some specific cases—i.e., when `ht` is

```
(* The type of our symbolic memories with hash-consing *)
Record hsmem:= {hpre: list hterm; hpost: PDict.t hterm}.

(* implementation of the [smem_valid] predicate *)
Definition hsmem_valid ge (hd: hsmem) (m:mem): Prop :=
   ∀ ht, List.In ht hd.(hpre) → hterm_eval ge ht m <> None

(* implementation of the symbolic memory evaluation *)
Definition hsmem_post_eval ge (hd: hsmem) x (m:mem): option value :=
   match PDict.get hd.(hpost) x with
   | None ⇒ Some (m x)
   | Some ht ⇒ hterm_eval ge ht m
   end.

(* The data-refinement relation *)
Definition smem_model ge (d: smem) (hd:hsmem): Prop :=
  (∀ m, hsmem_valid ge hd m ↔ smem_valid ge d m)
 ∧ ∀ m x, smem_valid ge d m →
     hsmem_post_eval ge hd x m = term_eval ge (d.(post) x) m.
```

Figure 3.9: Data-Refinement of Symbolic Memories with Hash-Consed Terms

an input or a constant, we know that `ht` cannot fail. In these cases, we avoid adding it to `hd.(hpre)`. Second, when `ht` is structurally equal to `(Input x)`, the implementation removes `x` from the dictionary: in other words, an assignment such as "$x := y$"—where $y \mapsto$ (`Input` x) in the current symbolic memory—resets x as unassigned. There is much room for future work on improving the `hsmem_set` operation by, e.g., applying rewriting rules on terms.[8]

Finally, we define the symbolic execution that invokes these operations on each assignment of the block. It is straightforward to prove that `(bblock_hsmem p)` refines `(bblock_smem p)` from the correctness properties of Figure 3.10.

```
Definition bblock_hsmem: bblock → ?? hsmem :=...

Lemma bblock_hsmem_correct p:
 WHEN bblock_hsmem p ⇝ hd THEN ∀ ge, smem_model ge (bblock_smem p) hd.
```

The Main Function of the Simulation Test Let us now present the main function of the simulation test, called `bblock_simu_test`, and sketched[9] in Fig. 3.11. First, it creates two memoizing functions `hC_term` and `hC_list_term` (Fig. 3.8) from the generic factory `hCons` (see Section 3.3.2 for details). Then, it invokes the symbolic execution `bblock_hsmem` on each block. Notice that these two symbolic executions share the memoizing functions `hC_term` and `hC_list_term`, meaning that each term produced by one of the symbolic executions is represented by a unique pointer. The symbolic executions produce two symbolic memories `d1` and `d2`. We compare them using two auxiliary functions. Hence, `(assert_eq_PDict d1.(hpost) d2.(hpost))` checks whether each pseu-

[8]Our implementation of `hsmem_set` is actually able to apply some rewriting rules. But, this feature is still not used by our verified scheduler.

[9]The code of `bblock_simu_test` has been largely simplified, by omitting the complex machinery which is necessary to produce an understandable trace for CompCert developers in the event of a negative answer.

doregister is assigned to the *same* term w.r.t pointer equality in both symbolic memories. Finally, `(assert_list_incl d2.(hpre) d1.(hpre))` checks whether each term of `d2.(hpre)` is also present in `d1.(hpre)`: i.e. whether all potential failures of `d2` are potential failures of `d1`. This last auxiliary function is implemented by FVDP and a "theorems-for-free" technique in Figure 2.18.

3.4 Conclusion of this Chapter

This chapter shows the ultra-lightweight FVDP of a hash-consing mechanism with the {Impure} library. This hash-consing mechanism is itself used for the FVDP of realistic applications: instruction schedulers in CompCert. Its correctness proof relies only on the correctness of {Impure} wrt Coq extraction, and the model of OCaml pointer equality in Coq. It does not depend on other properties that our OCaml code provides: in particular, the correctness proof of our verifier does not assume that two isomorphic hash-consed data structures in existence at the same time are always allocated to the same place in memory (but, this property is of course important for the "performance" of the scheduling checker).

In order not to have to convert CompCert's code generation flow to the full monadic style of {Impure}, at some point, we unsafely cast our verified scheduler from the {Impure} monad into a pure function. We do not think that this weakness hides a real issue: even if an unexpected bug in some of our OCaml oracles makes them nondeterministic, we do not call the scheduler twice on the same code, so there is no absurd case where we could go to if two different calls gave different results. This is in line with similar implicit assumptions elsewhere in CompCert that oracles are deterministic: see Section 2.1.

```
(* initial symbolic memory *)
Definition hsmem_empty: hsmem := {| hpre:= nil ; hpost := PDict.empty |}.

Lemma hsmem_empty_correct ge: smem_model ge smem_empty hsmem_empty.

(* symbolic evaluation of the right-hand side of an assignment *)
Fixpoint exp_hterm (e: exp) (hd hod: hsmem): ?? hterm :=
  match e with
  | PReg x ⇒
     match PDict.get hd.(post) x with
     | None ⇒ hInput x  (* smart constructor for Input *)
     | Some ht ⇒ RET ht
     end
  | Op o le ⇒
     DO lt ↞ list_exp_hterm le hd hod;;
     hApp o lt (* smart constructor for App *)
  | Old e ⇒ exp_hterm e hod hod
  end
with list_exp_hterm (le: list_exp) (d od: hsmem): ?? list_term :=
 ...
Lemma exp_hterm_correct e hd hod:
  WHEN exp_hterm e hd hod ⇝ ht THEN ∀ ge od d m,
   smem_model ge d hd → smem_valid ge d m →
   smem_model ge od hod → smem_valid ge od m →
     hterm_eval ge ht m = term_eval ge (exp_term e d od) m.

(* effect of an assignment on the symbolic memory *)
Definition hsmem_set (hd:hsmem) x (ht:hterm): ?? hsmem :=
  (* a weak version w.r.t the actual implementation *)
  RET {| hpre:= ht::hd.(hpre); hpost:=PDict.set hd x ht |}.

Lemma hsmem_set_correct hd x ht:
  WHEN hsmem_set hd x ht ⇝ hd' THEN ∀ ge d t,
    smem_model ge d hd →
    (∀ m, smem_valid ge d m → hterm_eval ge ht m = term_eval ge t m) →
      smem_model ge (smem_set d x t) hd'.
```

Figure 3.10: Refinement of the Operations of Figure 3.5 for Symbolic Memories with Hash-Consing

```
Definition assert_eq_PDict: ∀ {A}, PDict.t A → PDict.t A → ?? unit.
Lemma assert_eq_PDict_correct A (d1 d2 : PDict.t A):
  WHEN PDict.eq_test d1 d2 ⇝ _ THEN ∀ x, PDict.get d1 x = PDict.get d2 x.

Definition bblock_simu_test (p1 p2: bblock): ?? unit :=
  DO hC_term ↞ hCons {|hash_eq:=term_hash_eq; hashing:=term_hashing;
                       set_hid:=term_set_hid|};;
  DO hC_list_term ↞ hCons ... (* a record of type [(hashP list_hterm)] *)
  DO d1 ↞ bblock_hsmem hC_term hC_list_term p1;;
  DO d2 ↞ bblock_hsmem hC_term hC_list_term p2;;
  assert_eq_PDict d1.(hpost) d2.(hpost);;
  assert_list_incl d2.(hpre) d1.(hpre).

Theorem bblock_simu_test_correct (p1 p2 : bblock):
  WHEN bblock_simu_test p1 p2 ⇝ _ THEN bblock_simu p1 p2.
```

Figure 3.11: Verified Implementation of the AbstractBasicBlock Simulation Checker

Chapter 4

Polymorphic Factory Style for FVDP of an Abstract Domain of Polyhedra[†]

[We] are not so concerned with checking or generating proofs as with ***performing*** *proofs. Thus, we don't normally store [...] proofs but only the results of them - i.e.* ***theorems****. These form an* ***abstract type*** *on which the only allowed operations are the inference rules [...]; this ensures that a well-typed program cannot perform faulty proofs [...].*

The principal aims then in designing ML *were to make it impossible to prove non-theorems yet to program strategies for performing proofs.*

Mike Gordon et al. in "*A Metalanguage for Interactive Proof in* LCF" [Gor+78].

It is worth exhibiting one simple change among those which led [...] to the metalanguage ML*. [...] The simple change is not to index the proof by natural numbers, but instead to bind theorems to* ML *variables.*

Robin Milner in "LCF*: A Way of Doing Proofs with a Machine*" [Mil79].

Contents

[†]This chapter details some parts of [#Bou+18] and of [Mar17].

Polymorphic LCF style (also named "Polymorphic Factory Style", abbreviated as PFS) may be applied to various kind of computations, and not only to decision procedures as it could be suggested by Section 2.4.2. Indeed, PFS was proposed during the PhD of Maréchal [Mar17] in order to simplify the design of oracles of the {VPL} abstract domain—a formally verified abstract domain of convex polyhedra.

This chapter provides a tutorial on PFS oracles, illustrated on some operators of the {VPL}. It compares several FVDP designs and explains why PFS is the best one: it significantly simplifies the development and debugging of untrusted and formally verified components, while also reducing their running times. Section 4.1 is a very short introduction to the {VPL}, which defers to Chapter 7 the application of the {VPL} to formally verified static analysis. Section 4.2 uses the projection of convex polyhedra as an introductory running example of PFS oracle. Built on the top of this example, Section 4.3 provides an advanced example of PFS design for the convex hull of two convex polyhedra.

4.1 FVDP of an Abstract Domain of Polyhedra

We consider the formal verification of static analyzers—like Verasco [Jou+15]—, that aim at ensuring absence of *runtime errors* such as division by zero or invalid memory access in an input source program. The correctness of Verasco has been formally proved in Coq, i.e. if the analysis of a given C program does not raise any alarm, then this program cannot have any *undefined behavior*.

In abstract interpretation [CC77], the analyzer attaches to each program point an *invariant*, which is a property satisfied by all reachable states at this point. These invariants belong to classes of predicates called *abstract domains* that must provide operators for computing the disjunction of two invariants (`join`), their conjunction (`meet`), and the existential quantification of a variable in an invariant (`proj`). They must also provide tests for implication between invariants (`is_included`) and unsatisfiability (`is_empty`). The formal correctness of the analysis boils down to ensuring that all these operators compute overapproximations w.r.t. the concrete semantics (e.g. `join` computes an overapproximation of the disjunction). In particular, we do not need to formally prove that operators are precise (i.e. they compute tight results), even though they are in practice.

In the following, we focus on the abstract domain of convex polyhedra on $\mathbb{Q}$ [CH78], which is able to handle linear relations between numerical variables $\boldsymbol{x} \triangleq (x_1, \ldots, x_n) \in \mathbb{Q}^n$. For simplicity, we do not consider integer or floating point variables in this document. A *convex polyhedron* is a conjunction of linear constraints of the form $\sum_i a_i x_i \bowtie b$ where a_i, b are constants in $\mathbb{Q}$ and $\bowtie$ is $\geq$, $>$ or $=$. A polyhedron is represented as a list of `Cstr.t`, which is the type of linear constraints.[1] A lot of libraries feature polyhedral calculus, but only a few certify their results. The Coq-Polyhedra library [AK17] follows the autarkic approach; according to its authors, the goal of this library is not to perform efficient computations, but to formalize a large part of the convex polyhedra theory by using reflexive proofs. Fouilhé, Monniaux, and Périn [FMP13] initiated the Verified Polyhedra Library ({VPL}), an abstract domain for Verasco, in a FVDP design. Unlike most polyhedra libraries, {VPL} uses the constraints-only representation of polyhedra in order to ease its certification in Coq.[2] {VPL} certification also relies on witnesses that captures the information needed to prove the correctness of the polyhedral operators. Fortunately, proving their correctness reduces to verifying implications[3]

[1] As we only deal with convex polyhedra, the adjective convex is often omitted in the rest of the document.

[2] Most polyhedra libraries maintain a double representation of polyhedra as *constraints* and as *generators*, i.e. vertices and rays. Certifying them would require to prove the correctness of Chernikova's conversion algorithm. Instead, Fouilhé looked for efficient polyhedra operators in constraints-only representation.

[3] Note that a polyhedral implication $P_1 \Rightarrow P_2$ is geometrically an inclusion between polyhedra $P_1 \subseteq P_2$.

between polyhedra, in conjunction with other simple verifications that depend on the operator. For example, polyhedron P is empty iff $P \Rightarrow P_\emptyset$, where $P_\emptyset$ is a single contradictory constant constraint such as $0 \geq 1$. The emptiness of $P_\emptyset$ is thus itself checkable by a simple rational comparison.

Farkas's lemma gives a simple way to prove polyhedral implications [Far02]. It states that any nonnegative linear combination of the constraints of a polyhedron P is an obvious logical consequence of P. For instance, $x \geq 3 \wedge y \geq 0$ implies $2 \cdot (x \geq 3) + 1 \cdot (y \geq 0) = 2x + y \geq 6$, meaning that any point satisfying $x \geq 3 \wedge y \geq 0$ also satisfies $2x + y \geq 6$. Moreover, Farkas's lemma states that any polyhedral implication can be proved thanks to such simple computations on constraints and thus provides a theoretical foundation for designing the witness format of polyhedral operators [Bes+07; Bes+10]. The formulation below is restricted to polyhedra with non strict inequalities only, and Section 4.3.1 will provide a generalization to polyhedra with equalities and strict inequalities.

Lemma 4.1 (Farkas 1902)**.** *Let P_1 and P_2 be two polyhedra containing only nonstrict inequalities. Let us call* Farkas combination *of P_1 any nonnegative linear combination of P_1's constraints and of the trivial tautology $1 \geq 0$.*
Any Farkas combination *of P_1 is a logical consequence of P_1. Moreover, if $P_1 \Rightarrow P_2$ then*

- *either P_1 is empty and there exists a Farkas combination of P_1 producing the contradictory constraint $0 \geq 1$,*
- *or each constraint of P_2 is a Farkas combination of P_1.*

For instance, the polyhedron $x \geq 3 \wedge y \geq 0 \wedge -2x - y \geq -5$ is empty, as shown by the combination $2 \cdot (x \geq 3) + 1 \cdot (y \geq 0) + 1 \cdot (-2x - y \geq -5) = 0 \geq 1$. In general, efficiently finding the right combination relies on a Linear Programming (LP) solver [Chv83].

4.2 A Tutorial on PFS through the Projection of Convex Polyhedra

This section provides a tutorial on PFS oracles, using operator `proj` of the abstract domain of polyhedra as a running example. Indeed, this operator is at the heart of the {VPL}, since many {VPL} operators (`join`, `is_empty`, `is_included`, etc) can be derived from this one. See [Mar17].

The `proj` operator performs the elimination of existential quantifiers on polyhedra: given a polyhedron P and a variable x, (`proj` P x) computes a polyhedron P' such that $P' \Leftrightarrow \exists x, P$. Let us consider the example of Figure 4.1. Predicate P_0 expresses that q is the result of the Euclidean division of x by 3, with r as remainder. Predicate P_1 "instantiates" P_0 with $x = 15$. Then, predicate P'_1 corresponds to the computation of $\exists r, P_1$ (as a polyhedron on $\mathbb{Q}$).

$$P_0 \triangleq \begin{cases} & x = 3 \cdot q + r & [C_1] \\ \wedge & r \geq 0 & [C_2] \\ \wedge & r < 3 & [C_3] \end{cases} \qquad P'_1 \triangleq \begin{cases} & x - 15 = 0 & [C'_1] \\ \wedge & q - 4 > 0 & [C'_2] \\ \wedge & 5 - q \geq 0 & [C'_3] \end{cases}$$

$$P_1 \triangleq P_0 \wedge \; x = 15 \qquad [C_4]$$

Figure 4.1: Computation of P'_1 as "`proj` P_1 r"

Geometrically, (`proj` P x) represents the orthogonal projection of a polyhedron P according to direction x. The standard algorithm for computing this projection is Fourier-Motzkin elimination [Fou27]. Ongoing research is trying to improve efficiency with alternate algorithms [HK12; MMP17]. But in

our two-tier approach, the correctness proof of `proj` does not need to consider these implementation details.

We assume that for proving the correctness of our surrounding software (typically, a static analyzer), we do not need to prove $P' \Leftrightarrow \exists x, P$ but only $(\exists x, P) \Rightarrow P'$.[4] Thus, we only want to prove the correctness of `proj` as defined below.

Definition 4.1 (Correctness of `proj`)**.** Function `proj` is correct iff any result P' for a computation (`proj` $P\ x$) satisfies $(P \Rightarrow P') \ \wedge \ x \notin V(P')$ where $V(P')$ is the set of variables appearing in P' with a non-null coefficient.

The condition $x \notin V(P')$ ensures that variable x is no longer bounded in P'. As dynamic checking of this condition is fast and easy, we only look for a way to build P' from P which ensures by construction that $P \Rightarrow P'$. For this purpose, we exploit Farkas's lemma as follows. Internally, we handle constraints in the form "$t \bowtie 0$" where t is a linear term and $\bowtie \in \{=, \geq, >\}$. Hence, each input constraint "$t_1 \bowtie t_2$" is first normalized as "$t_1 - t_2 \bowtie 0$". Then, we generate new constraints using only the two operations of Definition 4.2. Obviously, such constraints are necessarily implied by P.

Definition 4.2 (Linear Combinations of Constraints)**.** We define operations + and $\cdot$ on normalized constraints by

- $(t_1 \bowtie_1 0) + (t_2 \bowtie_2 0) \triangleq (t_1 + t_2) \bowtie 0$
  where $\bowtie \triangleq \max(\bowtie_1, \bowtie_2)$ for the total increasing order induced by the sequence $=, \geq, >$.
- $n \cdot (t \bowtie 0) \triangleq (n \cdot t) \bowtie 0$
  under preconditions $n \in \mathbb{Q}$ and, if $\bowtie \in \{\geq, >\}$ then $n > 0$.

For example, P'_1 is generated from P_1 by the script on the right hand-side. Here `tmp` is an auxiliary constraint, where variable x has been eliminated from C_1 by rewriting using equality C_4.

$$
\begin{array}{rcl}
\texttt{tmp} & \leftarrow & C_4 + -1 \cdot C_1 \\
C'_1 & \leftarrow & C_4 \\
C'_2 & \leftarrow & \frac{1}{3} \cdot (C_3 + \texttt{tmp}) \\
C'_3 & \leftarrow & \frac{1}{3} \cdot (C_2 + -1 \cdot \texttt{tmp})
\end{array}
$$

In the following, we study how to design – in OCaml– a certified frontend `Front.proj` that monitors Farkas combinations produced by an untrusted backend `Back.proj`. Section 4.2.5 will then formalize `Front.proj` in Coq.

4.2.1 Naive but Unsound LCF Style

In a first step, we follow a naive LCF style. We thus consider two datatypes for constraints: modules `BackCstr` and `FrontCstr` define respectively the representation of constraints for the backend and the frontend.

Each module is accessed both in the backend and in the frontend, but the frontend representation is abstract for the backend. Hence, the visible interface of `FrontCstr` for the backend is given on the right-hand side. Type `Rat.t` represents set $\mathbb{Q}$, and `add` and `mul` represent respectively operators + and $\cdot$ on constraints.

```
module FrontCstr: sig
  type t
  val add: t -> t -> t
  val mul: Rat.t -> t -> t
end
```

When preconditions of "$\cdot$" are not satisfied, `mul` either raises an exception or returns a trivially satisfied constraint like "$0 = 0$".

[4] This may not be sufficient in other applications. See [CL21] for a counterexample.

Going back to our example, P'_1 is first computed from P_1 using backend constraints. Indeed, with its own representation, the backend finds the solution by efficient computations, combining complex datastructures, GMP rationals and even floating-point values. On the contrary, the frontend representation is based on certified code extracted from Coq. In particular, it uses internally the certified rationals of the Coq standard library, where integers are represented as lists of bits. Once a solution is found, the backend thus rebuilds this solution in the frontend representation. The easiest way is to make `Back.proj` compute the certified constraints (of type `FrontCstr.t`) in parallel with its own computations. Hence, we propose a first version of `Back.proj`, called `Back.proj0`, with the following type.

```
Back.proj0: (BackCstr.t * FrontCstr.t) list -> Var.t -> FrontCstr.t list
```

Let us define two functions:

1. a certified function `occurs: Var.t -> FrontCstr.t -> bool` such that `occurs x c` tests whether `x` $\in V($`c`$)$;
2. an untrusted function `export: FrontCstr.t -> BackCstr.t` that converts a frontend constraint into a backend one.

Then, we implement `Front.proj` as follows:

```
let Front.proj (p: FrontCstr.t list) (x: Var.t): FrontCstr.t list =
  let bp = List.map (fun c -> (export c, c)) p in
  let p' = List.map snd (Back.proj0 bp x) in
  if List.exists (occurs x) p'
  then failwith "oracle error"
  else p'
```

`Front.proj` only dynamically checks that $x \notin P'$. In particular, it does not verify that $P \Rightarrow P'$ holds, because it should follow directly from the correctness of `FrontCstr.add` and `FrontCstr.mul`. Ideally – mimicking a LCF style prover – function `Back.proj0` uses type `FrontCstr.t` as a type of theorems. It derives logical consequences of a list of constraints (of type `FrontCstr.t`) by combining them with `FrontCstr.mul` and `FrontCstr.add`. Like in a LCF style prover, there is no explicit "proof object" as value of this theorem type.

Unfortunately, this approach is unsound. We now provide an example which only involves two input polyhedra that are reduced to a single constant constraint. Let us imagine an oracle wrapping function `memofst` given below. Assuming that it is first applied to the unsatisfiable constraint $0 \geq 1$, this first call returns $0 \geq 1$, which is a correct answer. However, when it is then applied to the satisfiable constraint $2 \geq 0$, this second call still returns $0 \geq 1$, which is now incorrect! This unsoundness is severe, because even a faithful programmer could, by mistake, implement such a behavior while handling mutable data structures.

```
let memofst:FrontCstr.t -> FrontCstr.t =
  let first = ref None in
  fun c ->
    match !first with
    | None -> (first := Some c); c
    | Some c' -> c'
```

4.2.2 Generating an Intermediate Certificate

In order to avoid the unsoundness issue of the naive LCF style, we could instead introduce an intermediate data structure representing a trace of the backend computation. Then, the frontend would use

this trace to rebuild the certified result using its own certified data structures. Such a trace has the form of an Abstract Syntax Tree (AST) and is called a *certificate*. This approach was used to design the first version of the {VPL} [FMP13; #FB14]. In the following, we detail the process of certificate generation and why we prefer avoiding it.

We define below a certificate type named `pexp`. It represents a type of polyhedral computations, and depends on type `fexp` that corresponds to Farkas combinations. Constraints are identified by an integer. Type `pexp` provides a `Bind` construct for computing auxiliary constraints like `tmp` in the example of P'_1.

```
type fexp =
  | Ident of int
  | Add of fexp * fexp
  | Mul of Rat.t * fexp
```

```
type pexp =
  | Bind of int * fexp * pexp
  | Return of fexp list
```

Figure 4.2 gives an example of certificate for P'_1, where each input constraint C_i is represented by "`Ident` i". The intermediate constraint `tmp` is bound to identifier 5.

```
Bind (5, Add (Ident 4, Mul (−1, Ident 1)),
      Return [ Ident 4;
               Mul (1/3, Add (Ident 3, Ident 5));
               Mul (1/3, Add (Ident 2, Mul (−1, Ident 5))) ])
```

Figure 4.2: A certificate for P'_1

Next, we easily implement in Coq a `Front.run` interpreter of `pexp` certificates—in the sense of Figure 1.2(b)—and prove that it only outputs a logical consequence of its input polyhedron.

```
Front.run: pexp -> (FrontCstr.t list) -> (FrontCstr.t list)
```

Let us mention that when a `pexp` uses certificate identifiers that have no meaning w.r.t. `Front.run`, this latter fails. For the following, we do not need to specify how identifiers are generated and attached to constraints. We leave this implementation detail underspecified.

Now, we need to turn `Back.proj0` into a function `Back.proj1` where each `BackCstr.t` constraint in input is associated to a unique identifier.

```
Back.proj1: (BackCstr.t * int) list -> Var.t -> pexp
```

However, `Back.proj1` is more complex to program and debug than `Back.proj0`. Indeed, in LCF style, certified operations run in "parallel" with the oracle. On an oracle bug (for instance, if the oracle multiplies an inequality by a negative scalar), the LCF style checker raises an error right at the point where the bug appears in the oracle: this makes debugging of oracles much easier. In contrast, in presence of an ill-formed certificate, the developer has to find out where does the ill-formness comes from in its oracle. Moreover, an oracle like `Back.proj1` needs to handle constraint identifiers for `Bind` according to their semantics in `Front.run`. As detailed in Section 4.3, this is particularly painful on Fouilhé's implementation of the `join` operator, because it involves several spaces of constraint names (one for each "implication proof"). In the following, we present two solutions that fix the issue of naive LCF style.

4.2.3 Standard LCF Style

Intuitively, the lying function `memofst` of Section 4.2.1 exploits the fact that constraints of the result P' are typed with a single type of "theorems", whereas these "theorems" are relative to a given set of axioms: the input constraints of P. The standard LCF style fixes this issue by memorizing in the type of "theorems" the set of axioms in which these theorems have been derived. In other words, in standard LCF style, a Farkas combination is encoded by a sequent "$P \vdash C$" where P is a polyhedron and C a constraint: P is the polyhedron to which the Farkas combination is applied and C is the result of the Farkas combination. This enables the front-end to dynamically check that oracles do not mix these sequents in an unsound way.

Figure 4.3 sketches a standard LCF style implementation in OCaml. For the sake of simplicity, this implementation uses `BackCstr.t` as an internal representation of constraints: a sequent "$P \vdash C$" is a encoded as a pair (P, C) where P is a list of constraints and C a constraint. This implementation thus wraps operations of `BackCstr` module, but with defensive verifications ensuring that such a pair (P, C) always satisfies the invariant "$P \Rightarrow C$".

```
module FrontCstr: sig

  (* restricted interface for Backend computations *)
  type t
  val add: t -> t -> t
  val mul: Rat.t -> t -> t

  (* extended interface for Frontend computations *)
  val import: BackCstr.t list -> t list
  val export: t list -> BackCstr.t list
  ...
```

```
module FrontCstr = struct

  type t = BackCstr.t list * BackCstr.t
  let add (p1,c1) (p2,c2) =
      assert (p1 == p2); (p1, BackCstr.add c1 c2)
  let mul r (p,c) =
      assert ((BackCtr.is_eq c) || (Rat.is_positive r));
      (p, BackCstr.mul r c)

  let import p = List.map (fun c -> (p,c)) p
  let export p = List.map snd p
  ...
```

Figure 4.3: Interface and (Sound) Implementation of `FrontCstr` in Standard LCF Style

In standard LCF style, an oracle can still use the `memofst` function. But this will be detected at runtime and rejected by the frontend. As explained below, Polymorphic Factory Style (PFS) improves this by preventing the cheating `memofst` at compile-time (with static typechecking). Moreover it makes the implementation of the front-end even more lightweight, since some defensive checks are removed.

4.2.4 Polymorphic Factory Style

The principle of PFS is very simple: instead of abstracting the "type of theorems" (i.e. the type `FrontCstr.t`) using an ML abstract datatype, we abstract it using ML polymorphism. As explained above, the lying function `memofst` of Section 4.2.1 exploits the fact that we have a *static* type of theorems, whereas when we interpret constraints of the result P' as theorems, they are relative to the input constraints of P. Hence, this issue would disappear by using instead a *dynamic* type, generated at each call to the oracle. Using ML polymorphism, we actually express that our oracle is parameterized by any of such dynamic type of theorems.

In practice, the type `FrontCstr.t` used in backend oracles – e.g. `Back.proj` – is replaced by `'c`. In order to allow the backend to build new "theorems" – i.e. Farkas combinations – we introduce a polymorphic record type `fLCF` (acronym of "*farkas Logical Consequences Factory*").

```
type 'c fLCF = {
  add: 'c -> 'c -> 'c;
  mul: Rat.t -> 'c -> 'c
}
```

Then, the previous oracle `Back.proj0` that we defined for the simple LCF style is generalized into

```
val Back.proj: 'c fLCF -> (BackCstr.t * 'c) list -> Var.t -> 'c list
```

Intuitively, function `Back.proj0` could now be redefined as:

```
(Back.proj add=FrontCstr.add; mul=FrontCstr.mul)
```

Let us point out here that the type of `Back.proj` implementation must generalize this signature, and not simply unify with it. This directly forbids the `memofst` bug: if we remove the type coercion from the code of `memofst`, the type system infers `memofst: '_a -> '_a` where `'_a` is an existential type variable introduced for a sound typing of references [Wri95; Gar02]. Hence, a cheating use of `memofst` would prevent oracles (like `Back.proj`) from having an acceptable type.

In other words, the unsoundness of `memofst` is *statically* detected, at compile-time. This is a first significant advantage over standard LCF style, where it was only detected at runtime. Moreover, in PFS, Farkas combinations do not need to track the list of axioms P, because this is only necessary for the defensive checks of standard LCF style. PFS is slightly simpler and more efficient than standard LCF style. This is a second advantage over standard LCF style. Beyond these two advantages, Section 4.3 and Chapter 6 illustrate that the polymorphic style provides interesting opportunities to reuse oracles for free, whereas, in the style based on type abstraction, this would require a refactorization of oracles with explicit functors.

4.2.5 Formalizing `proj` Frontend in Coq

Let us now sketch how the frontend is formalized in Coq: the technique generalizes the one of Section 2.4.2. We define the type `Var.t` as `positive` – the Coq type for binary positive integers. We build the module `FrontCstr` of constraints encoded as radix trees over `positive` with values in `Qc`, which is the Coq type for $\mathbb{Q}$. Besides operations `add` and `mul`, module `FrontCstr` provides two predicates: `(sat c m)` expresses that a model `m` satisfies the constraint `c`; and `(noccurs x c)` expresses that variable `x` does not occur in constraint `c`.

```
sat: t → (Var.t → Qc) → Prop.
noccurs: Var.t → t → Prop.
```

We also prove that `sat` is preserved by functions `add` and `mul`. Then, these predicates are lifted to polyhedra `p` of type `(list FrontCstr.t)`.

```
Definition sat p m := List.Forall (fun c ⇒ FrontCstr.sat c m) p.
Definition noccurs x p := List.Forall (FrontCstr.noccurs x) p.
```

Because `front_proj` invokes a nondeterministic computation (the external oracle as detailed below), it is itself a nondeterministic computation. Here is its type and its specification:

```
front_proj: list FrontCstr.t → Var.t → ??(list FrontCstr.t).
Lemma front_proj_correctness: ∀ p x p',
   (front_proj p x) ⇝ p' → (∀ m, sat p m → sat p' m) ∧ noccurs x p'.
```

We implement `front_proj` in PFS, as explained in Section 4.2.4. First, we declare a `fLCF` record type containing operations for frontend constraints. These operations do not need to be declared as nondeterministic: in the Coq frontend, they will be only instantiated by pure Coq functions. Then, `back_proj` is defined as a nondeterministic computation. The type of `back_proj` is given uncurried in order to avoid nested "??" type transformers. At extraction, this axiom is replaced by a wrapper of `Back.proj` from Section 4.2.4.

```
Record fLCF A := { add: A → A → A; mul: Qc → A → A }.
Axiom back_proj: ∀ {A},
   ((fLCF A) * (list (FrontCstr.t * A))) * Var.t → ??(list A).
```

Like in Section 4.2.4, `back_proj` receives each constraint in two representations: an opaque one of polymorphic type `A` and a transparent one of another type. For simplicity, this document uses `FrontCstr.t` as the transparent representation on the Coq side. [5]

Now, let us sketch how we exploit our polymorphic `back_proj` to implement `front_proj` and prove its correctness. For a given `p: (list FrontCstr.t)`, parameter `A` of `back_proj` is instantiated with `wcstr(sat p)` where `wcstr(s)` is the type of constraints satisfied by any model satisfying `s`. In other words, `wcstr(sat p)` is the type of logical consequences of `p`, i.e. the type of its Farkas combinations. Hence, instantiating parameter `A` of `back_proj` by this dependent type expresses that combinations from the input `p` and from the `fLCF` operations are satisfied by models of `p`. Concretely, `(front_proj p x)` binds the result of `(back_proj ((mkInput p), x))` to a polyhedron `p'` and checks that `x` does not occur in `p'`.

```
Record wcstr(s: (Var.t → Qc) → Prop) :=
  { rep: FrontCstr.t; rep_sat: ∀ m, s m → FrontCstr.sat rep m }.

mkInput: ∀ p, fLCF(wcstr(sat p)) * list(FrontCstr.t * wcstr(sat p)).
```

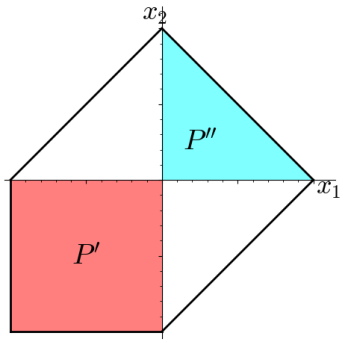
Actually, `rep_sat` above can be seen as a data-invariant attached to a `rep` value. This invariant is trivially satisfied on the input values, i.e. the constraints of `p`. And, it is preserved by `fLCF` operations. These two properties are reflected in the type of `mkInput`. The polymorphism of `back_proj` is a way to ensure that `back_proj` preserves any data-invariant like this one, on the output values (see Section 2.2.3).

[5] In order to avoid unnecessary conversions from `FrontCstr.t` to `BackCstr.t` (that would be hidden in `back_proj` wrapper), our actual implementation uses instead an axiomatized type which is replaced by "`BackCstr.t`" at extraction: this is similar to the implementation of Fouilhé and Boulmé [#FB14].

4.3 FVDP of Polyhedra Convex-Hull

This section illustrates an advanced usage of polymorphic factories through the convex hull of convex polyhedra—i.e. the `join` operator for the abstract domain. It gives another nontrivial example of a "correct by construction" oracle. It also illustrates the flexible power of PFS, by deriving `join` from the projection oracle of Section 4.2.4. On this `join` oracle, PFS induces a drastic simplification w.r.t to the first version of the {VPL} by removing many cumbersome rewritings on certificates. Indeed, we simply derive the certification of the `join` operator by invoking the projection operator on a *direct product* of factories. As we detail below, such a product computes two independent polyhedral inclusions, in parallel.

In abstract interpretation, `join` approximates the disjunction of two invariants. For the abstract domain of polyhedra, this disjunction geometrically corresponds to the union of two polyhedra $P' \cup P''$. However, in general, such a union is not a convex polyhedron. Operator `join` thus overapproximates this union by the convex hull $P' \sqcup P''$ that *we define* as the smallest convex polyhedron containing $P' \cup P''$. For instance, given



$$P' \triangleq \{x_1 \leq 0,\ x_2 \leq 0, x_1 \geq -1,\ x_2 \geq -1\}$$

$$P'' \triangleq \{x_1 \geq 0,\ x_2 \geq 0, x_1 + x_2 \leq 1\}$$

$P' \sqcup P'' \triangleq \{x_1 \geq -1,\ x_2 \geq -1,\ x_1{+}x_2 \leq 1,\ x_2{-}x_1 \geq -1,\ x_2{-}x_1 \leq 1\}$ as represented on the right hand side figure as the black outline.

The correctness of `join`, given in Definition 4.3, is reduced to two implications themselves proved by Farkas' lemma. More precisely, on a computation ($\texttt{join}\ P'\ P''$), the oracle produces internally two lists of Farkas combinations that build a pair of polyhedra (P_1, P_2) satisfying $P' \Rightarrow P_1$ and $P'' \Rightarrow P_2$. Then, the frontend checks that P_1 and P_2 are syntactically equal. If the check is successful, it returns polyhedron P_1.

Definition 4.3 (Correctness of `join`)**.** Function `join` is correct iff any result P of a computation ($\texttt{join}\ P'\ P''$) satisfies $(P' \Rightarrow P) \ \wedge\ (P'' \Rightarrow P)$.

4.3.1 Extended Farkas Factories

The factory operations of Definition 4.2 are sufficient to compute any result of a projection, but they do not suffice for the convex hull and more generally for proving all kinds of polyhedra inclusions. The definition 4.4 given here completes this set of operations. Lemma 4.2 ensures its completeness for proving polyhedra inclusions. It extends Lemma 4.1 for polyhedra with equalities and strict inequalities.

Definition 4.4 (Extended Farkas Combination)**.** An extended Farkas combination may invoke one of the five operations:

- $(t_1 \bowtie_1 0) + (t_2 \bowtie_2 0) \triangleq (t_1 + t_2) \bowtie 0$
  where $\bowtie \triangleq \max(\bowtie_1, \bowtie_2)$ for the total increasing order induced by the sequence $=, \geq, >$.
- $\begin{cases} n \cdot (t \bowtie 0) \triangleq (n \cdot t) \bowtie 0 \text{ under preconditions } n \in \mathbb{Q} \text{ and, if } \bowtie \in \{\geq, >\} \text{ then } n > 0 \\ 0 \cdot (t \bowtie 0) \triangleq (0 = 0) \end{cases}$

- `weaken`$((t \bowtie 0)) \triangleq (t \geq 0)$, for all linear terms t and $\bowtie \in \{=, \geq, >\}$.
- `cte`$(n, \bowtie) \triangleq (n \bowtie 0)$ assuming $n \in \mathbb{Q}$ and $n \bowtie 0$.
- `merge`$((t \geq 0), (-t \geq 0)) \triangleq (t = 0)$, for all linear terms t.

Besides the operations of Definition 4.4, we also define $\top$ as a shortcut for `cte`$(0, =)$ that thus corresponds to constraint $0 = 0$. Hence, $\top$ is neutral for operations $+$ and $\cdot$ on constraints. It is thus a very convenient default value in our oracles.

Lemma 4.2 (Extended Farkas Lemma)**.** *Let P_1 and P_2 be two convex polyhedra on $\mathbb{Q}$ such that $P_1 \Rightarrow P_2$. Then,*

- *either P_1 is empty and a contradictory constant constraint (e.g. $0 > 0$) is a Farkas combination of P_1,*
- *or each constraint of P_2 is an extended Farkas combination of P_1.*

From now on, we only consider extended Farkas combinations and omit the adjective "extended". Definition 4.4 leads to extend our factory type as given on the right-hand side.
Field `top` corresponds to $\top$. Type `cmpT` is our enumerated type of comparisons representing $\{\geq, >, =\}$.

```
type 'c fLCF =
{ top: 'c;
  add: 'c -> 'c -> 'c;
  mul: Rat.t -> 'c -> 'c;
  weaken: 'c -> 'c;
  cte: Rat.t -> cmpT -> 'c;
  merge: 'c -> 'c -> 'c }
```

4.3.2 Encoding `join` as a Projection

Most polyhedra libraries use the double representation of polyhedra, as constraints and as generators. Computing the convex hull $P' \sqcup P''$ using generators is easy. It consists in computing the union of generators and in removing the redundant ones. In constraints-only, the convex hull is computed as a projection problem, following the algorithm of Benoy, King, and Mesnard [BKM05]. The convex hull is the set of convex combinations of points from P' and P'', i.e.

$$\{\boldsymbol{x} \mid \boldsymbol{x}' \in P',\ \boldsymbol{x}'' \in P'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''\} \tag{4.1}$$

To express that a point belongs to a polyhedron in a more computational way, we introduce the following matrix notation. We denote $\boldsymbol{x}' \in P'$ by $\boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}'$, where each line of this system represents one constraint of P'. Similarly, $\boldsymbol{x}'' \in P''$ is rewritten into $\boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}''$. The previous set of points (4.1) becomes

$$\{\boldsymbol{x} \mid \boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}',\ \boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''\} \tag{4.2}$$

Then, by eliminating variables α', α'', $\boldsymbol{x}'$ and $\boldsymbol{x}''$, we obtain $P' \sqcup P''$. Note that we cannot directly use operator `proj` to compute this projection because the set of points (4.2) is defined with a nonlinear constraint $\boldsymbol{x} = \alpha' \cdot \boldsymbol{x}' + \alpha'' \cdot \boldsymbol{x}''$. We go back to linear constraints by applying the changes of variable $\boldsymbol{y}' := \alpha' \cdot \boldsymbol{x}'$ and $\boldsymbol{y}'' := \alpha'' \cdot \boldsymbol{x}''$. By multiplying matrix $\boldsymbol{A}'\boldsymbol{x}' \geq \boldsymbol{b}'$ by α' and $\boldsymbol{A}''\boldsymbol{x}'' \geq \boldsymbol{b}''$ by α'', we obtain equivalent systems $\boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}'$ and $\boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}''$. The set of points (4.2) is now described as $\{\boldsymbol{x} \mid P_H\}$ where P_H is the conjunction of linear constraints:

$$P_H \triangleq \{\boldsymbol{A}'\boldsymbol{y}' \geq \alpha' \cdot \boldsymbol{b}',\ \boldsymbol{A}''\boldsymbol{y}'' \geq \alpha'' \cdot \boldsymbol{b}'',\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1,\ \boldsymbol{x} = \boldsymbol{y}' + \boldsymbol{y}''\} \tag{4.3}$$

For our previous example where $\boldsymbol{x} \triangleq (x_1, x_2)$, the polyhedron P_H is:

$$\begin{cases} -y'_1 \geq 0,\ -y'_2 \geq 0,\ y'_1 \geq -\alpha',\ y'_2 \geq -\alpha' & (\boldsymbol{A'y'} \geq \alpha' \cdot \boldsymbol{b'}) \\ y''_1 \geq 0,\ y''_2 \geq 0,\ -y''_1 - y''_2 \geq -\alpha'' & (\boldsymbol{A''y''} \geq \alpha'' \cdot \boldsymbol{b''}) \\ \alpha' \geq 0,\ \alpha'' \geq 0,\ \alpha' + \alpha'' = 1 & \\ x_1 = y'_1 + y''_1,\ x_2 = y'_2 + y''_2 & \text{(Encoding convex combinations)} \end{cases}$$

Operator `join` finally consists in eliminating variables α', α'', $\boldsymbol{y'}$ and $\boldsymbol{y''}$ from P_H. The presence of equalities or strict inequalities requires an additional pass that follows the projection, involving operators `weaken` and `merge` of the factory. We omit this step in the document in order to keep our explanations simple. Moreover, in practice, encoding P_H of Equation (4.3) could be done more efficiently by considering fewer variables, exploiting the fact that $\alpha'' = 1 - \alpha'$ and $\boldsymbol{y''} = \boldsymbol{x} - \boldsymbol{y'}$. But as this complicates the understanding and does not affect much the certification, this document will not consider this improvement.

We now compare certificate style to PFS for proving `join` from results of `proj`. For the sake of simplicity, we consider here only the case where polyhedra do not contain strict inequalities.

4.3.3 Proving `join` with Certificates

As previously explained about Definition 4.3, the correctness of `join` is ensured by building the convex hull P from two Farkas combinations, one of P' and one of P''. Fouilhé, Monniaux, and Périn [FMP13] described how to extract such combinations from the result of the projection of P_H. As in the rest of the polyhedra library they developed, they use certificates-producing oracles. Thus, their `join` has the following type:

```
Back.join1 : (BackCstr.t * int) list -> (BackCstr.t * int) list ->
             pexp * pexp
```

It takes the two polyhedra P' and P'' as input, and each of their constraints is attached to a unique identifier, as explained in Section 4.2.2. It returns two certificates of type `pexp`, one for each inclusion $P' \Rightarrow P$ and $P'' \Rightarrow P$ of Definition 4.3.

Let us now detail how Fouilhé, Monniaux, and Périn [FMP13] retrieve such certificates from the projection of P_H. Consider operator `Back.proj1_list` that extends `Back.proj1` from Section 4.2 by eliminating several variables one after the other instead of a single one. Assume that `Back.proj1_list`—when applied on polyhedron P_H and variables α', α'', $\boldsymbol{y'}$, $\boldsymbol{y''}$—returns $(P, \boldsymbol{\Lambda})$ where $\boldsymbol{\Lambda}$ is a certificate of type `pexp` showing that $P_H \Rightarrow P$.

Recall that certificate $\boldsymbol{\Lambda}$ corresponds to a constructive proof that computes each constraint of P by a Farkas combination of constraints P_H. Thus, $\boldsymbol{\Lambda}$ remains a valid certificate of $\sigma(P_H) \Rightarrow \sigma(P)$ for any substitution σ on the variable space of P_H. Moreover, since variables α', α'', $\boldsymbol{y'}$, $\boldsymbol{y''}$ have been eliminated in P, then P is a fixpoint of any substitution σ with a domain included in this set of variables: for such a σ, we have $\sigma(P) = P$. The key idea is thus to find well-chosen substitutions of α', α'', $\boldsymbol{y'}$, $\boldsymbol{y''}$ in order to retrieve certificates for $P' \Rightarrow P$ and $P'' \Rightarrow P$ out of $\boldsymbol{\Lambda}$.

Indeed, recall that P_H represents the set of convex combinations $\alpha' \cdot \boldsymbol{x'} + \alpha'' \cdot \boldsymbol{x''}$ of points $\boldsymbol{x'} \in P'$ and $\boldsymbol{x''} \in P''$. By setting $\alpha' \mapsto 1$ and $\alpha'' \mapsto 0$, P_H becomes restricted to P'. More precisely, considering substitution $\sigma_1 \triangleq (\alpha' \mapsto 1,\ \alpha'' \mapsto 0,\ \boldsymbol{y'} \mapsto \boldsymbol{x},\ \boldsymbol{y''} \mapsto \boldsymbol{0})$ and after normalizing $\sigma_1(\boldsymbol{x} = \boldsymbol{y'} + \boldsymbol{y''})$ into $\boldsymbol{0} = \boldsymbol{0}$, we get:

$$\sigma_1(P_H) = \{\boldsymbol{A'x} \geq \boldsymbol{b'},\ \boldsymbol{0} \geq \boldsymbol{0},\ 1 \geq 0,\ 0 \geq 0,\ 1 + 0 = 1,\ \boldsymbol{0} = \boldsymbol{0}\} \tag{4.4}$$

Hence, $\sigma_1(P_H)$ syntactically "extends" P' (i.e. $\boldsymbol{A'x} \geq \boldsymbol{b'}$) with trivial tautologies. We are thus able to find a certificate of $P' \Rightarrow P$ by a kind of "partial application" of $\boldsymbol{\Lambda}$ to these trivial tautologies. In other

words, the certificate of $P' \Rightarrow P$ is a simple "inlining" in $\mathbf{\Lambda}$ of the constraints of $\sigma_1(P_H)$ that are not physically present in P'.

The same reasoning applied with substitution $\sigma_2 \triangleq (\alpha' \mapsto 0,\ \alpha'' \mapsto 1,\ \boldsymbol{y}' \mapsto \mathbf{0},\ \boldsymbol{y}'' \mapsto \boldsymbol{x})$ leads to find a certificate for $P'' \Rightarrow P$.

4.3.4 Proving `join` with a Direct Product of Polymorphic Farkas Factories

In PFS, we use the following type for `join`'s oracle:

```
Back.join : 'c1 fLCF -> (BackCstr.t * 'c1) list ->
            'c2 fLCF -> (BackCstr.t * 'c2) list ->
            ('c1 -> 'c2 -> 'c3) -> 'c3 list
```

In this polymorphic type, variable `'c1` (resp. `'c2`) represents the type of logical consequences of P' (resp. P''), whereas variable `'c3` represents the type of logical consequences of $P' \cup P''$, i.e. the type of constraints that are *both* logical consequences of P' and P'' (see Definition 4.3). The `Back.join` oracle is parametrized by a certified operator (given by the frontend) of type `'c1 -> 'c2 -> 'c3` and called `unify`. This `unify` operator simply tests whether the two input constraints are syntactically equal: in this case, this constraint is trivially a logical consequence of $P' \cup P''$. Otherwise, `unify` *fails*: it returns a `top` constraint (or raises an error). Hence, `Back.join` builds a convex polyhedron P which includes—*by construction*—the set $P' \cup P''$.

Internally, this oracle first builds a pair of polyhedra (P_1, P_2) of type `('c1 list)*('c2 list)` and then computes P by pairwise applying `unify` to these two lists. Hence, alternatively to a result of type `('c1 -> 'c2 -> 'c3) -> 'c3 list`, we could also design `Back.join` for a result of type `('c1 list)*('c2 list)`, and let the frontend build P from this result. These two alternatives are more or less equivalent, because building P from the pair (P_1, P_2) is very easy to implement and prove correct in Coq.[6]

We said that for computing the convex hull, `join` eliminates variables α', α'', $\boldsymbol{y}'$ and $\boldsymbol{y}''$ from P_H. Recall that the projection operator that we defined for PFS in Section 4.2.4 has type

```
Back.proj: 'c fLCF -> (BackCstr.t * 'c) list -> Var.t -> 'c list
```

As we did for the certificate approach, let us define `Back.proj_list` that extends `Back.proj` by eliminating a list of variables.

```
Back.proj_list: 'c fLCF -> (BackCstr.t * 'c) list -> Var.t list -> 'c list
```

The `Back.join` oracle is parametrized by two Farkas factories, but needs to call `Back.proj_list` on a single one. To do so, `Back.join` will provide `Back.proj_list` with a combination of its own two factories. Although the parameter `'c fLCF` of `Back.proj_list` was originally designed to be provided by the frontend, nothing forbids the backend from tuning it. This is where the flexibility of PFS comes into play! More precisely, `Back.join` combines the two factories `f1:'c1 fLCF` and `f2: 'c2 fLCF` into a new one of type `('c1 * 'c2) fLCF` given in Figure 4.4. This factory computes with frontend constraints from P' and P'' in parallel: it corresponds to the *direct product* of the two initial Farkas factories.

Now, let us detail how `Back.join` builds polyhedron P_H in input of `Back.proj_list`. Each constraint of P_H must be represented as a tuple `(BackCstr.t * ('c1 * 'c2))`. In particular, it must be attached to a pair of frontend constraints of type `'c1 * 'c2`. As explained in Section 4.3.3,

[6] Introducing `unify` is conceptually interesting when generalizing the approach for constraints with equalities or strict inequalities. For example, `unify` may unify two constraints $x = 3$ and $x > 3$ into $x \geq 3$.

```
let factory_product (f1: 'c1 fLCF) (f2: 'c2 fLCF): ('c1 * 'c2) fLCF =
{
  top = (f1.top, f2.top);
  add  = (fun (c1,c2) (c1',c2') -> (f1.add c1 c1', f2.add c2 c2'));
  mul = (fun r (c,c') -> (f1.mul r c, f2.mul r c'));
  weaken = (fun (c,c') -> (f1.weaken c, f2.weaken c'));
  cte = (fun r cmp -> (f1.cte r cmp, f2.cte r cmp));
  merge = (fun (c1,c1') (c2,c2') ->  (f1.merge c1 c1', f2.merge c2 c2'));
}
```

Figure 4.4: Direct product of two Farkas Factories

our proof of $P' \Rightarrow P$ (resp. $P'' \Rightarrow P$) corresponds to "instantiate" the proof of $P_H \Rightarrow P$ returned by `Back.proj_list` into a proof of $\sigma_1(P_H) \Rightarrow P$ (resp. $\sigma_2(P_H) \Rightarrow P$). Thus, we only need to attach each constraint C of P_H to a pair $(\sigma_1(C), \sigma_2(C))$ of type `'c1 * 'c2`. Remark that constant constraints—like "$0 \geq 0$" and "$1 \geq 0$" of Equation (4.4)—are easily built in type `'c1` (resp. `'c2`) thanks to operators `f1.cte` (resp. `f2.cte`).

In other words, we embed in P_H constraints of P'—of type `(BackCstr.t * 'c1)`—from the input of `Back.join`, by attaching them to constraint "$0 \geq 0$" of type `'c2`. Indeed, recalling that P' informally written $\boldsymbol{A'x} \geq \alpha'$ should be embedded in P_H as $\boldsymbol{A'y'} \geq \alpha'\boldsymbol{b'}$, we get:

$$\boldsymbol{A'y'} \geq \alpha'\boldsymbol{b'} \left\{ \begin{array}{cccccc} \texttt{BackCstr.t} & * & (& \texttt{'c1} & * & \texttt{'c2} \quad) \\ \boldsymbol{A'_1y'} \geq \alpha'b'_1 & , & (& \underbrace{\sigma_1(\boldsymbol{A'_1y'} \geq \alpha'b'_1)}_{\boldsymbol{A'_1x} \geq b'_1} & , & \underbrace{\sigma_2(\boldsymbol{A'_1y'} \geq \alpha'b'_1)}_{0 \geq 0} \quad) \\ \ldots & & & & & \\ \boldsymbol{A'_p y'} \geq \alpha'b'_p & , & (& \underbrace{\sigma_1\left(\boldsymbol{A'_p y'} \geq \alpha'b'_p\right)}_{\boldsymbol{A'_p x} \geq b'_p} & , & \underbrace{\sigma_2\left(\boldsymbol{A'_p y'} \geq \alpha'b'_p\right)}_{0 \geq 0} \quad) \end{array} \right.$$

Similarly, we embed constraints of P''—of type `(BackCstr.t * 'c2)`—by attaching them to constraint "$0 \geq 0$" of type `'c1`:[7]

$$\boldsymbol{A''y''} \geq \alpha''\boldsymbol{b''} \left\{ \begin{array}{cccccc} \boldsymbol{A''_1y''} \geq \alpha''b''_1 & , & (& \underbrace{\sigma_1(\boldsymbol{A''_1y''} \geq \alpha''b''_1)}_{0 \geq 0} & , & \underbrace{\sigma_2(\boldsymbol{A''_1y''} \geq \alpha''b''_1)}_{\boldsymbol{A''_1x} \geq b''_1} \quad) \\ \ldots & & & & & \\ \boldsymbol{A''_q y''} \geq \alpha''b''_q & , & (& \underbrace{\sigma_1\left(\boldsymbol{A''_q y''} \geq \alpha''b''_q\right)}_{0 \geq 0} & , & \underbrace{\sigma_2\left(\boldsymbol{A''_q y''} \geq \alpha''b''_q\right)}_{\boldsymbol{A''_q x} \geq b''_q} \quad) \end{array} \right.$$

Then, we embed constraints $\alpha' \geq 0$ and $\alpha'' \geq 0$ in type `BackCstr.t * ('c1 * 'c2)` with:

$$\begin{array}{cccccc} \texttt{BackCstr.t} & * & (& \texttt{'c1} & * & \texttt{'c2} \quad) \\ \alpha' \geq 0 & , & (& \underbrace{\sigma_1(\alpha' \geq 0)}_{1 \geq 0} & , & \underbrace{\sigma_2(\alpha' \geq 0)}_{0 \geq 0} \quad) \\ \alpha'' \geq 0 & , & (& \underbrace{\sigma_1(\alpha'' \geq 0)}_{0 \geq 0} & , & \underbrace{\sigma_2(\alpha'' \geq 0)}_{1 \geq 0} \quad) \end{array}$$

At last, constraints $\alpha' + \alpha'' = 1$ and of $\boldsymbol{x} = \boldsymbol{y'} + \boldsymbol{y''}$ are all represented in type `('c1 * 'c2)` by `(f1.top, f2.top)`.

[7] When generalizing the approach for constraints with equalities or strict inequalities, we replace $\sigma_2(\boldsymbol{A'y'} \bowtie \alpha'\boldsymbol{b'})$ and $\sigma_1(\boldsymbol{A''y''} \bowtie \alpha''\boldsymbol{b''})$ by $\top$ instead of $\boldsymbol{0} \bowtie \boldsymbol{0}$ (which is UNSAT if $\bowtie$ is strict inequality). Indeed, informally, $\boldsymbol{A'y'} \bowtie \alpha'\boldsymbol{b'}$ results from $\alpha' \cdot (\boldsymbol{A'x'} \bowtie \boldsymbol{b'})$ with $\boldsymbol{y'} = \alpha'.\boldsymbol{x'}$. Thus, with this view, assigning α' to 0 simplifies $\boldsymbol{A'y'} \bowtie \alpha'\boldsymbol{b'}$ into $\top$.

For our example of convex hull, let us focus on the proof that P' and P'' both imply $-x_1 - x_2 \geq -1$ (i.e. $x_1 + x_2 \leq 1$), which is a constraint of $P' \sqcup P''$. We build P_H as described above, and obtain from its projection a pair of frontend constraints, that is

$$(-x_1 \geq 0,\ 0 \geq 0) + (-x_2 \geq 0,\ 0 \geq 0) + (1 \geq 0,\ 0 \geq 0) + (0 \geq 0,\ -x_1 - x_2 \geq -1)$$

$$= (-x_1 - x_2 \geq -1,\ -x_1 - x_2 \geq -1)$$

Above, the left-hand side of each pair of constraint is a frontend constraint of type `'c1`, and the right hand side is of type `'c2`. From P' point of view, we obtain $-x_1 - x_2 \geq -1$ as the combination of $-x_1 \geq 0$ (i.e. $x_1 \leq 0$), $-x_2 \geq 0$ (i.e. $x_2 \leq 0$) and the constant constraint $1 \geq 0$ that comes from $\alpha' \geq 0$. On the other hand, $-x_1 - x_2 \geq -1$ is a constraint of P'' and is directly returned as a frontend constraint of type `'c2`. The projection returns such results for each constraint of the convex hull $P' \sqcup P''$.

In conclusion, with a well chosen factory, we define our PFS `join` as a simple call to `proj_list`. This makes our implementation much simpler (and more efficient) than Fouilhé's one, where the two certificates of `join` are obtained from the one of `Back.proj1_list` by tedious rewritings involving three name spaces of constraint identifiers (one for the projection, and two for the input polyhedra).

Chapter 5

Polymorphic Factory Style for Certifying Answers of Boolean SAT-Solvers

[We] performed extensive experiments on benchmarks from the 2016 SAT competition and the 2015 SAT race. [...] In total, there were 381,468,814 lines in the 225 proofs totalling 250 GByte [...]

The Coq *checker verified 161 out of these 225 instances within a maximum runtime of 24 hours. For the remaining 64 instances, it timed out (59), ran out of memory (1), or determined that the proof was invalid (4). [...]*

The ACL2 *checker verified 212 out of the 225 instances within a maximum runtime of 6,708 seconds [slightly less than 2 hours], typically being at least an order of magnitude faster than the* Coq *checker. For the remaining 13 instances, it ran out of memory (1), crashed (1), or determined that the proofs were invalid (11).*

Luís Cruz-Filipe et al. in "*Efficient Certified RAT Verification*" [Cru+].

Contents

This chapter refines the toy example of Section 2.4.2 into to a realistic use-case: {SatAnsCert}, a verifier of SAT-solver answers, itself certified in Coq. Actually, for verifying UNSAT answers, we were inspired by a previous Coq development, called "lrat checker", documented in [Cru+] and available online[1]. Our main contribution is to illustrate how our PFS helps to develop a code that

[1] https://imada.sdu.dk/~petersk/lrat/

is much more scalable than this previous one—for a very modest development effort.[2] Actually, this case study is challenging for PFS: in contrast to examples of previous chapters, the intermediate RAT clauses [WHJ13] generated by recent SAT-solvers on UNSAT answers may *not be logical consequences* of the input CNF. This makes the design of our UNSAT verifier more subtle.

5.1 Overview of {SatAnsCert} and its formal correctness

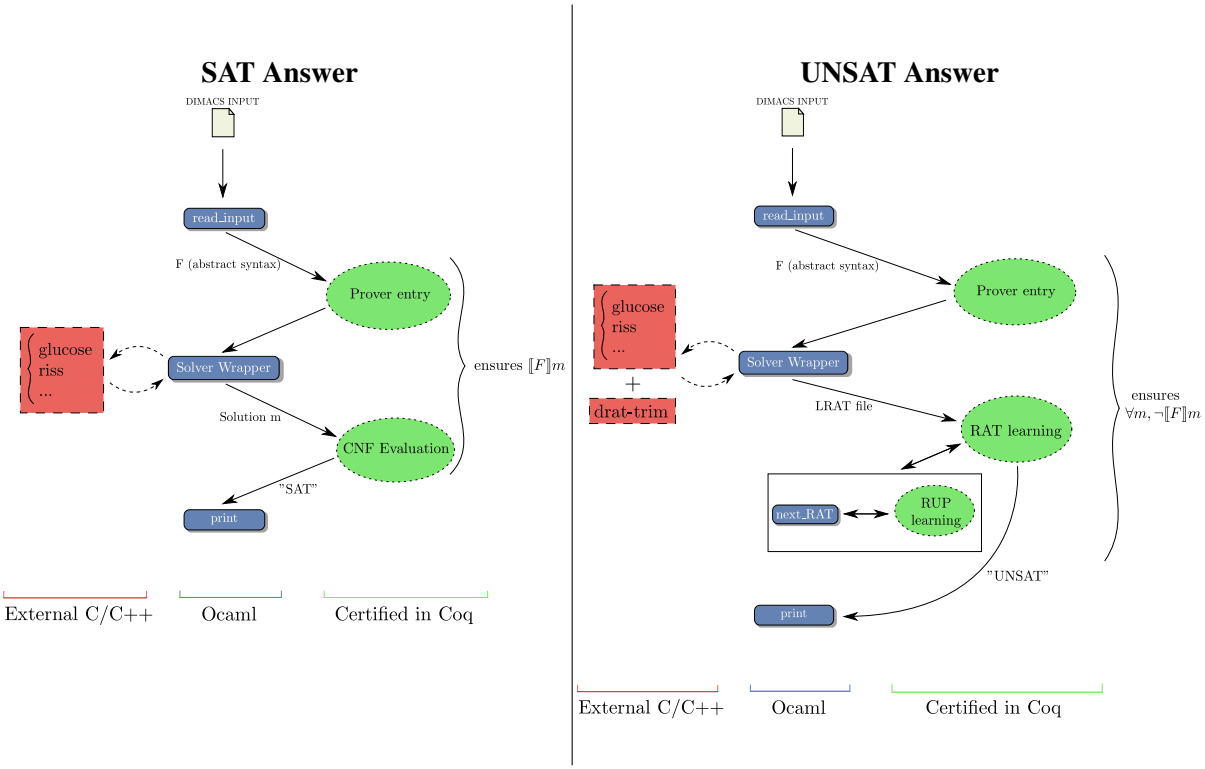


Figure 5.1: Overview of {SatAnsCert}

{SatAnsCert} reads a proposition f in Conjunctive Normal Form and outputs whether f is *"SAT"* or *"UNSAT"* (see Definition 2.5). This proposition f must be syntactically given in DIMACS file – a standard format[3] of SAT competitions. Internally, {SatAnsCert} invokes – according to options on its command line – some state-of-the-art-in-2018 SAT-solver like Glucose[4], Riss[5], CryptoMinisat[6] or CaDiCaL[7]. This SAT-solver is expected to produce a witness of its answer (such a witness is mandatory for SAT competitions since 2016). {SatAnsCert} thus checks this witness before outputting the answer or failing on an error. The execution of {SatAnsCert} is depicted in Figure 5.1. The external SAT-solver is run in a separate process and communicates with {SatAnsCert} through the file system. As later detailed, {SatAnsCert} also invokes some OCaml oracles through the FFI of the

[2] The full Coq/OCaml code of these examples is online at https://github.com/boulme/satans-cert.
[3] https://www.satcompetition.org/2009/format-benchmarks2009.html
[4] http://www.labri.fr/perso/lsimon/glucose
[5] http://tools.computational-logic.org/content/riss.php
[6] https://github.com/msoos/cryptominisat
[7] http://fmv.jku.at/cadical

```
1  Program Definition main: ?? unit :=
2    TRY
3      DO f ⇜ read_input();; (* Command-line + CNF parsing *)
4      DO a ⇜ sat_solver f;; (* solver(+drat-trim) wrapper *)
5      match a with
6      | SAT_Answer mc ⇒
7        assert_b (satProver f mc) "wrong SAT model";;
8        ASSERT (∃ m, ⟦f⟧m);;
9        println "SAT !"
10     | UNSAT_Answer ⇒
11       unsatProver f;;
12       ASSERT (∀ m, ¬⟦f⟧m);;
13       println "UNSAT !"
14   WITH_ANY e ⇒
15     DO s ⇜ exn2string e;;
16     println ("Certification failure: " +; s).
```

Figure 5.2: (simplified) Coq code of the main function of {SatAnsCert}

{Impure} library (presented in Chapter 2): these oracles are thus part of the {SatAnsCert} process. The external SAT-solver is actually invoked through one of these OCaml oracles.

We now describe the formal property proved on {SatAnsCert} in Coq+Impure+OCaml. First, like in CompCert, I/O (i.e. parsing and printing) are not formally proved and thus must be trusted. More precisely, the formal correctness of {SatAnsCert} only deals with the abstract syntax (defined in Figure 2.19 page 45) of the input CNF. And it is directly expressed in the main function of {SatAnsCert} through statically proved "`ASSERT`" (see Figure 5.2). Here, "`ASSERT P`" (where `P : Prop`) is simply a macro for "`RET (A:=P) _`": it declares a proof of proposition `P` that must be (statically) provided as a proof obligation generated by "`Program Definition`". We consider that the ability to use imperative code in Coq with statically verified assertions improves the approach of CompCert– where formally proved components and unproved trusted components are linked together in OCaml only.

Hence, our code in Figure 5.2, thus combines *static* assertions ("`ASSERT`") and *dynamic* assertions, like "`assert_b`" defined on page 35. The static "`ASSERT`" proved at line 8 derives from the defensive check of line 7: `satProver` simply evaluates CNF `f` in the model `mc` found by the SAT-solver. Similarly, the static "`ASSERT`" proved at line 13 derives from a defensive check of line 12: `unsatProver` checks that the UNSAT witness (here implicit) provided by the SAT-solver is valid, or fails otherwise.

In summary, the actual TCB of {SatAnsCert} corresponds to around 20 Coq lines for the specification of CNF (Figure 2.19) + 40 Coq lines for the main of (Figure 5.2) + 80 lines of OcamlLex (parsing of input Dimacs file) + 100 lines of OCaml (implem. of `read_input`) + {Impure} (and Coq, OCaml, OcamlLex).

The next sections sketch how verification of UNSAT answers is achieved. Section 5.3 defines a first simple version with type:

```
unsatProver (f: cnf): ?? (∀ m, ¬⟦f⟧m)
```

And Section 5.4 defines a second refined version with the equivalent type:

```
unsatProver (f: cnf): ?? ¬(∃ m, ⟦f⟧m)
```

5.2 Certifying UNSAT answers of SAT-solvers: a brief overview

Since the pioneering works of [GN03] and [ZM03], the verification of UNSAT answers has been well studied. Several proof formats have been proposed, and currently, the DRAT format [WHJ14; Heu16] is the standard format in SAT competitions. Actually, most SAT-solvers generate only DRUP proofs [Gel08; HJW13a] – a previous format that DRAT has later extended with RAT clauses [WHJ13]. In theory, using RAT clauses may lead to exponentially shorter proofs than using only pure (D)RUP proofs. But, in practice, the SAT-solving community is still looking for efficient algorithms to find such RAT proofs [HKB17; Kie+20].

5.2.1 Background on RUP proofs and CDCL (Conflict-Driven Clause Learning)

In a first step, we will focus on (D)RUP proofs: they are simpler to understand. Actually, we even consider a subset of RUP proofs, introduced as "*restricted RUP* proofs" in [CMS17], that I rename (for clarity) into "*backward resolution* proofs". Indeed, I find clearer to present this proof system as a variant of the *resolution proof system* where the resolution rule is *specialized* for backward reasoning through the rule BckRsl of Definition 5.1 below.

Definition 5.1 (Backward Resolution Chain)**.** Given these two clause derivation rules,

$$\textsc{BckRsl}\ \frac{c_1 \qquad \{\neg\ell\} \cup c_2}{c_2}\ c_1 \backslash c_2 = \{\ell\} \qquad\qquad \textsc{Triv}\ \frac{c_1}{c_2}\ c_1 \backslash c_2 = \emptyset$$

for $n \geq 1$, we write "$c_1, \ldots, c_n \vdash^{\text{BRC}} c$" **iff** there is a bottom-up derivation – like on the right hand-side – that first iterates BckRsl from c on the list $c_1, \ldots, c_{n-1}$ and then concludes by Triv on c_n.

$$\textsc{BckRsl}\ \frac{c_1 \qquad \textsc{BckRsl}\ \dfrac{c_{n-1} \qquad \textsc{Triv}\ \dfrac{c_n}{\ldots}}{\ldots}}{c}$$

When "$f \vdash^{\text{BRC}} c$", we say that f is a *Backward Resolution Chain* (BRC) of c.

The usual resolution rule can be replaced by backward resolution chains: for all literal ℓ, all clauses c_1 and c_2, we have $\{\ell\} \cup c_1, \{\neg\ell\} \cup c_2 \vdash^{\text{BRC}} c_1 \cup c_2$ or $\{\ell\} \cup c_1 \vdash^{\text{BRC}} c_1 \cup c_2$. Indeed, if $\ell \notin c_1 \cup c_2$, then the left alternative holds because $(\{\ell\} \cup c_1) \backslash (c_1 \cup c_2) = \{\ell\}$ (BckRsl) and $(\{\neg\ell\} \cup c_2) \backslash (\{\neg\ell\} \cup c_1 \cup c_2) = \emptyset$ (Triv).[8] Otherwise, the second alternative holds by Triv rule.

As a consequence of the above remark, the correctness & completeness of the resolution proof system (see Theorem 2.1 of page 44) is rephrased by Theorem 5.1.

Theorem 5.1 (Correctness & completeness of backward resolution proofs)**.** *A CNF f is UNSAT* ***iff*** *there exists a sequence of clauses $c_1, \ldots, c_n$ (with $n \geq 1$) such that*

- *forall $i \in [1, n]$, there exists a list of clauses $f_i \subseteq f \cup \{c_1, ..., c_{i-1}\}$ such that $f_i \vdash^{\text{BRC}} c_i$*
- *and, $c_n = \emptyset$*

Such a sequence $c_1, \ldots, c_n$ is called a RUP proof *of the unsatisfiability of f.*

[8] In other words, if $\ell \notin c_1 \cup c_2$, the usual resolution rule is derived by

$$\textsc{BckRsl}\ \frac{\{\ell\} \cup c_1 \qquad \textsc{Triv}\ \dfrac{\{\neg\ell\} \cup c_2}{\{\neg\ell\} \cup c_1 \cup c_2}}{c_1 \cup c_2}$$

DIMACS file	**DRUP file**	**LRAT file**

`p cnf 3 5`
`1 2 3 0`
`-1 2 0`
`-2 3 0`
`2 -3 0`
`-2 -3 0`
	`-2 0`
	`0`
		`6 -2 0 3 5 0`
		`7 0 6 4 2 1 0`

Input CNF	**Learned (RUP) Clauses**	**BRC for each RUP Clause**
$c_1: \{1,2,3\}$ $c_2: \{-1,2\}$ $c_3: \{-2,3\}$ $c_4: \{2,-3\}$ $c_5: \{-2,-3\}$	$\{-2\}$ $\emptyset$	$\dfrac{\dfrac{}{c_5}\,\textsc{Triv} \quad \dots}{c_6 : \{-2\}}\,\textsc{BckRsl}$ $\dfrac{\dfrac{\dfrac{}{c_1}\,\textsc{Triv} \quad \dots}{c_2}\,\textsc{BckRsl} \quad \dots}{c_4}\,\textsc{BckRsl} \quad \dots}{c_7 : \emptyset}\,\textsc{BckRsl}$

Figure 5.3: Example of DIMACS/DRUP/LRAT files associated to their meaning below. In these files, each positive names either a Boolean variable or a clause (depending on the context); "0" marks the end of a sequence. Note that there is a unique and easy way to fill the "..." in the backward resolution chains: this is left to the formally verified checker of LRAT files.

For example, Figure 5.3 refutes the input CNF with a RUP proof reduced to a sequence of two clauses: $\{-2\}$ and then $\emptyset$. The corresponding BRCs appear on the right hand-side of the figure.

Now, we sketch how RUP proofs are naturally found by CDCL SAT-solvers, a refinement of DPLL algorithms, at the heart of modern SAT-solvers (see [SLM09] for details). Rule Triv corresponds to the fact that, under its side-condition, the proposition "$c_1 \wedge \neg c_2$" is UNSAT. Hence, Triv corresponds exactly to a *conflict* on clause c_1 where "$\neg c_2$" represents the current assignment of literals (i.e. as a conjunction of literals). Similarly, the BckRsl rule corresponds to the fact that, under its side-condition, the proposition "$c_1 \wedge \neg c_2$" implies the proposition "$\{\ell\} \wedge \neg c_2$", the latter being equivalent to "$\neg(\neg\{\ell\} \vee c_2)$". Actually, this corresponds exactly in DPLL SAT-solving to a *unit-propagation* on clause c_1 where "$\neg c_2$" represents the assignment of literals before the propagation and "$\neg(\{\neg\ell\} \cup c_2)$" represents the assignment after the propagation (i.e. where ℓ has been assigned to "TRUE"). In other words, a BRC exactly corresponds to a sequence of unit-propagations (BckRsl rule) leading to a conflict (Triv rule). The paradigm of CDCL SAT-solvers is to *learn* lemmas (under assumption of the input CNF) from conflicts: each of these lemmas is actually a clause provable from a BRC involving the input clauses and previously learned clauses. The solver answers "UNSAT", when it has learned the empty clause: the sequence of its learned clauses is then exactly a RUP proof (RUP is the acronym of "*Reverse Unit Propagation*").

5.2.2 Checking DRUP proofs

Historically, some CDCL SAT-solvers have dumped full resolution proofs on UNSAT answers (see [ZM03]). As illustrated by Section 2.4.2, certifying a resolution proof checker is not too difficult

and, in Coq, a first checker has been certified by [Arm+10]. However, instrumenting SAT-solvers to output full resolution proofs is very intrusive. Thus, RUP proofs have been proposed as a very lightweight alternative for the design of SAT-solvers [Gel08]. In counterpart, checking RUP proofs requires recovering all BRCs, typically by replaying unit-propagations. In practice, a RUP-checker does not need all the heuristics of a CDCL SAT-solver, but the data-structures necessary for unit-propagation (e.g. *two-watched literals* [Gen13]).

The DRUP proof format [HJW13a] is an ASCII file format to describe a RUP proof as a list of clauses, one by line (see Figure 5.3). There are also lines to delete clauses which are no longer involved in remaining resolution chains.[9] The standard checker of DRUP proofs in SAT competitions is currently DRAT-trim[10] of [WHJ14]. Of course, it also checks DRAT proofs, a conservative extension of DRUP with RAT clauses (detailed at Section 5.4).

Actually, checking DRUP/DRAT proofs is still a complex task (see [RC18]) and DRAT-trim is an untrusted program written in C. Hence, DRAT-trim has been designed to output the full BRC of learned clauses, in another proof format called LRAT (see Figure 5.3). As indicated by its name, DRAT-trim first prunes from the proof (by processing it backward) many learned clauses that are not necessary to derive the empty clause. This greatly reduces the size of LRAT proofs (and of DRAT-trim running times).

Then, [Cru+] have developed two certified checkers of LRAT proofs: one certified in Coq and extracted to OCaml; the other certified in ACL2 and extracted to C. As shown in Figure 5.4 – built from the benchmark table published by Peter Schneider-Kamp on his webpage[1] – their Coq/OCaml version is terribly slow compared to their ACL2/C version.

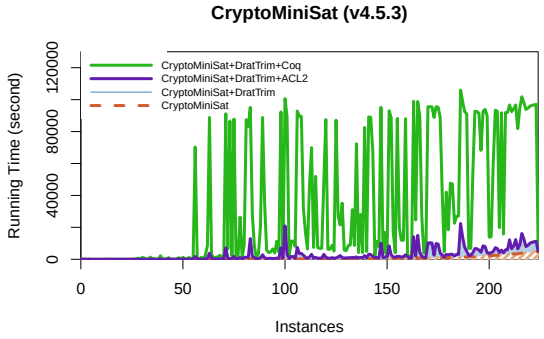


Fig. 5.4. Benchmark of [Cru+]

Our work illustrates that, by using the {Impure} library, we can improve the efficiency of the Coq/OCaml implementation, while substantially simplifying the Coq proof.

5.3 Verification of (D)RUP proofs in {SatAnsCert}

This section describes how `unsatProver` introduced at page 86 is implemented by checking the LRAT file generated with DRAT-trim from a DRUP proof (itself generated by the SAT-solver, as represented in Figure 5.1).

5.3.1 Polymorphic Factory Style of the untrusted LRAT Parser

We now introduce our shallow embedding of RUP proofs in Coq. This is slight variant of the embedding of resolution proofs, introduced in Section 2.4.2.

[9] Deletions of clauses are not illustrated in our examples, because they have no influence on the correctness of UNSAT proofs. However, considering them crucially improves performance in practice. In {SatAnsCert} approach, only untrused OCaml oracles need to consider them. Hence, the formally verified Coq proof does not even model them.

[10] Available at https://www.cs.utexas.edu/~marijn/drat-trim

Checking a Backward Resolution Chain is defined by the following function, called `learn` (it builds a new consequence of the set of models).

```
learn: ∀{s}, list(consc s) → iclause → ??(consc s)
```

It is implemented (for "performance" only) such that if `l` $\vdash^{\text{BRC}}$ `c` then `(learn l c)` returns `c'` where `(rep c')=c`. An exception is raised on an invalid BRC.

The `unsatProver` function needs to parse the LRAT file and to check that it corresponds to a valid RUP proof of the input CNF. It delegates the parsing of the LRAT file to an external *untrusted* OCaml oracle. This LRAT parser is declared in Coq by the `rup_lratParse` axiom (see below). Following the style of Section 2.4.2, this function is parametrized by:

- an abstract type of clause: this type – called `C` – is abstract for the untrusted parser but instantiated by "`consc`$[\![f]\!]$" in the Coq proof;
- a logical consequence factory of type "`(rupLCF C)`": this factory allows the oracle to build logical consequences (i.e. new abstract clauses) with a BRC from existing ones thanks to function `rup_learn` (instantiated by the previous `learn` in the Coq proof).
- the input CNF f given as a list abstract clauses of type `C`.

```
Record rupLCF C :=
  { rup_learn:(list C) → iclause → ?? C;  get_id: C → clause_id }.
Axiom rup_lratParse: ∀ {C}, (rupLCF C)*list(C) → ?? C.
```

By using the `get_id` function, the parser first builds a map from clause identifiers in the DIMACS input to their corresponding abstract clause (i.e. axiom). Then, it maintains this map while parsing the LRAT file, i.e. when deleting clauses or adding new learned clauses. On a non-RUP clause or on unexpected issues in the LRAT file, it raises an exception. Otherwise, it eventually returns an abstract clause, and the checker verifies that this clause is empty. Like in Section 2.4.2, `unsatProver` is simply defined by the code below.

```
Definition unsatProver f: ?? (∀ m, ¬⟦f⟧m) :=
  DO c ↜ rup_lratParse (mkInput f);; assertEmpty c.
```

The *Polymorphic LCF style* design of our RUP checker has the following benefits w.r.t. the design of the prover found in [CMS17] (a preliminary version of the Coq implementation of the LRAT prover of [Cru+]): BRCs are verified "on-the-fly" in the oracle, and this is much easier to debug; the dictionary mapping *clause identifiers* to *clause values* is only managed by the OCaml oracle (in an efficient hash-table); hence, the deletion of clauses from memory is also only managed by the oracle; the Coq code is thus very simple and very small.

5.4 Generalization to (D)RAT proofs

A RUP proof can be thought as a sequence of transformations on the input CNF: each learned clause is added to the CNF. These transformations preserves logical equivalence. The motivation of RAT clauses – introduced in [HJW13b; WHJ13] – is to allow transformations which may break logical equivalence but preserve satisfiability. This could dramatically reduce the size of the CNF, and thus the size of its potential UNSAT proof.

Example 5.1. Let us define two CNFs f_1 and f_2 over arbitrary literals $(l_i)_{i\in[1,n]}$ and $(l'_j)_{j\in[1,p]}$ and over a distinct variable x:

$$f_1 \;=\; \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{p}(l_i \vee l'_j) \qquad f_2 \;=\; (\bigwedge_{i=1}^{n}(\neg x \vee l_i)) \wedge \bigwedge_{j=1}^{p}(x \vee l'_j)$$

Whereas f_1 has $n \cdot p$ clauses (of two literals), f_2 has only $n + p$ clauses (of two literals). These two CNF are equisatisfiable, which is easy to check by rewriting each of them into an equivalent DNF:

$$f_1 \;\Leftrightarrow\; (\bigwedge_{i=1}^{n} l_i) \vee (\bigwedge_{j=1}^{p} l'_j) \qquad f_2 \;\Leftrightarrow\; (x \wedge \bigwedge_{i=1}^{n} l_i) \vee (\neg x \wedge \bigwedge_{j=1}^{p} l'_j)$$

But, f_1 and f_2 are generally not equivalent, because f_2 constrains x whereas f_1 does not.

While a RUP proof corresponds to the skeleton of a resolution proof, a RAT proof corresponds to the skeleton of a kind of extended resolution proof. Like in extended resolution, fresh Boolean variables may be introduced during the proof, which helps reduce the size of formulas.

5.4.1 Introduction to RAT bunches

In this section, following [Lam17a], we slightly generalize the definition of RAT clauses of [Cru+] by considering the learning at once of a "bunch" of several RAT clauses on the same pivot. We first need to reintroduce the notion of RUP clause originally defined by [Gel08].

Definition 5.2 (RUP clause)**.** Given a CNF f and a clause c, we say that "*c is RUP w.r.t f*" – and we write $f \vdash^{\text{RUP}} c$ – **iff** one of the two following conditions is verified:

1. there exists l such that $\{l, \neg l\} \vdash^{\text{BRC}} c$ (i.e. c is a trivial tautology)
2. or, there exists f' with $f' \subseteq f$ such that $f' \vdash^{\text{BRC}} c$.

It is obvious that "$f \vdash^{\text{RUP}} c$" implies "$f \Rightarrow c$".

Definition 5.3 (RAT bunch)**.** Given two CNFs f_1 and f_2 and a literal l, we say that f_2 is a *bunch of RAT clauses w.r.t. f_1 for pivot l* – and we write $f_1 \vdash_l^{\text{RAT}} f_2$ – **iff** for each clause $c_2 \in f_2$ the two following conditions are satisfied:

(1) $l \in c_2$; (2) $f_1 \vdash^{\text{RUP}} (c_1 \backslash \{\neg l\}) \cup c_2$ for each clause c_1 of f_1.

Lemma 5.1 (SAT preservation of RAT)**.** *Let us assume $f_1 \vdash_l^{\text{RAT}} f_2$ and $[\![f_1]\!]m$. Then, there exists m' such that $[\![f_1 \wedge f_2]\!]m'$.*

Proof. If $[\![f_2]\!]m$ then the property is trivially satisfied for $m' = m$. Otherwise, let m' be the model defined from m by assigning l to true. By condition (1), we have $[\![f_2]\!]m'$. Let $c_2 \in f_2$ such that $\neg[\![c_2]\!]m$. For all $c_1 \in f_1$, from $[\![f_1]\!]m$ and condition (2) we deduce that $[\![(c_1 \backslash \{\neg l\}) \cup c_2]\!]m$, and thus $[\![c_1 \backslash \{\neg l\}]\!]m$, and thus $[\![c_1]\!]m'$. Hence, we have also $[\![f_1]\!]m'$. □

Let us remark that if $c_1 = c_1 \backslash \{\neg l\}$ (i.e. $\neg l \notin c_1$) then condition (2) of Definition 5.3 is trivially satisfied. This leads to introduce the notion of "*basis*" by Definition 5.4 below. Indeed, it suffices to only check condition (2) on clauses c_1 that are in the basis of f_1 w.r.t. pivot l.

Definition 5.4 (Basis)**.** Given a CNF f_1 and a literal l, the *basis of f_1 w.r.t. pivot l* is defined as the set of clauses in f_1 containing $\neg l$.

Example 5.2 (RAT bunches of Example 5.1)**.** Clauses of f_2 are checked w.r.t f_1 in two RAT bunches:

1. $f_1 \vdash^{\mathrm{RAT}}_{\neg x} \bigwedge_{i=1}^{n}(\neg x \vee l_i)$: checking this RAT bunch is trivial because the basis is empty.
2. $f_1 \wedge \bigwedge_{i=1}^{n}(\neg x \vee l_i) \vdash^{\mathrm{RAT}}_{x} \bigwedge_{j=1}^{p}(x \vee l'_j)$: here the basis is $\bigwedge_{i=1}^{n}(\neg x \vee l_i)$. We simply check that for all $(i, j) \in [1, n] \times [1, p]$, we have $(l_i \vee l'_j) \vdash^{\mathrm{BRC}} (l_i \vee x \vee l'_j)$ with $(l_i \vee l'_j) \in f_1$.

From Theorem 5.1, we deduce that if f_1 is SAT then $f_1 \wedge f_2$ is also SAT, and finally that f_2 is SAT (deleting clauses also trivially preserves satisfiability).

The reasoning of Example 5.2 is instantiated in Figure 5.5 which also provides an example of LRAT file with RAT clauses. Figure 5.5 instantiates Example 5.2 with $x = 8$, and for $i \in \{1, 2, 3\}$, $l_i = i$ and $l'_i = i + 3$. In the input CNF, subformula $c_1 \wedge \ldots \wedge c_9$ corresponds to f_1 while subformula $c_{10} \wedge c_{11}$ is the negative normal form of the DNF of f_1 (given at Example 5.1). Hence, the LRAT proof starts by learning f_2 as $c_{12} \wedge \ldots \wedge c_{17}$, and then finds a contradiction by learning $\{-8\}$ and $\{8\}$ from $f_2 \wedge c_{10} \wedge c_{11}$.

Input CNF

c_1 : $\{1, 4\}$
c_2 : $\{1, 5\}$
c_3 : $\{1, 6\}$
c_4 : $\{2, 4\}$
c_5 : $\{2, 5\}$
c_6 : $\{2, 6\}$
c_7 : $\{3, 4\}$
c_8 : $\{3, 5\}$
c_9 : $\{3, 6\}$
c_{10} : $\{-1, -2, -3\}$
c_{11} : $\{-4, -5, -6\}$

LRAT file

```
12 -8 1 0 0
13 -8 2 0 0
14 -8 3 0 0
15 8 4 0 -12 1 -13 4 -14 7 0
16 8 5 0 -12 2 -13 5 -14 8 0
17 8 6 0 -12 3 -13 6 -14 9 0
18 -8 0 12 13 14 10 0
19 0 18 15 16 17 11 0
```

Interpretation

Bunch of pivot -8 $\begin{cases} c_{12} : \{-8, 1\} \\ c_{13} : \{-8, 2\} \\ c_{14} : \{-8, 3\} \end{cases}$
learned from empty basis.

Bunch of pivot 8 $\begin{cases} c_{15} : \{8, 4\} \\ c_{16} : \{8, 5\} \\ c_{17} : \{8, 6\} \end{cases}$
learned from basis c_{12}, c_{13}, c_{14}.

Conclude by learning RUP clauses $c_{18} : \{-8\}$ and then $c_{19} : \emptyset$.

Figure 5.5: Example of LRAT file combining RAT and RUP clauses. Each line starts with the definition of a learned clause ended by a first "`0`". Then, either we have a non-empty list of positive integers ended by "`0`" and this list is expected to provide the BRC proving the clause as RUP, or we are learning a RAT clause with a pivot given by its first literal. In this second case, the line continues with a (possibly empty) list of non-null integer sequences ended by "`0`": each of these sequences starts with a negative number—providing the name of a clause in the basis of the RAT—followed by a list of positive integers (e.g. a BRC proving condition (2) of Definition 5.3).

Example 5.3 (Contradictory RAT bunches)**.** Given x and y two distinct variables. We check the two following RAT bunches: $x \vdash^{\mathrm{RAT}}_{\neg y} \neg y$ and $x \vdash^{\mathrm{RAT}}_{y} y$. This check is trivial because the basis is empty in both cases.

This last example shows that two *contradictory* RAT clauses can be learned from the same satisfiable CNF. Hence, "learning" a RAT clause **is not like** "learning" a new lemma: "learning" a RAT clause **is like** adding an axiom which preserves consistency.

5.4.2 Formalization of RAT bunches

In the syntax of LRAT files, each RAT clause comes with a list of BRC, one for each clause of the basis (see Figure 5.5 for an example and [Cru+] for full details). Note that a valid BRC is at least

of length 1. Here, by convention, a BRC of length 0 simply encodes the case (1) of Definition 5.2 (trivial tautology). Moreover, when these lists of BRC share a common prefix, this prefix can be given separately. We reflect these syntactic informations of LRAT files in the following Coq structure: field `clause_to_learn` is the clause to learn, `propag` is the common prefix of the BRC, and `rup_proofs` is the list of suffix of the BRC (one by clause of the basis). Here type `C` represents the type of clauses that are logical consequences of the current CNF (like in Section 5.3.1).

```
Record RatSingle C: Type :=
 { clause_to_learn:iclause; propag:list C; rup_proofs:list(list C) }.
```

Learning a RAT bunch is defined in Coq by the function `learnRat` below. In this function, parameter `s` is the set of models of the current CNF. The bunch is given in field `bunch` of parameter `R` where `pivot` is the pivot and `basis` (resp. `rem` – for remainder) is a list of clauses *containing* (resp. *not containing*) the negation of the pivot. If f_2 is the list of clauses to learn in `bunch`, then `learnRat` either returns the CNF "`basis` $\wedge$ `rem` $\wedge$ f_2" or fails if it cannot prove that the `bunch` is a correct RAT bunch.

```
Record RatInput C: Type :=
 { pivot:literal; rem:list C; basis:list C; bunch:list(RatSingle C) }.
Definition learnRat {s:model → Prop} (R:RatInput (consc s)):??cnf := ...
Lemma learnRat_correct (s: model → Prop) (R: RatInput (consc s)):
  WHEN learnRat R ⇝ f THEN ∀ m, s m → ∃ m', ⟦f⟧ m'.
```

Example 5.4 (Learning RAT bunches of Example 5.2)**.** The running example can be turned into two successive formal invocations of `learnRat`:

1. On the first time, we learn CNF "$f_1 \wedge \bigwedge_{i=1}^{n}(\neg x \vee l_i)$" with the empty basis, with $\bigwedge_{i=1}^{n}(\neg x \vee l_i)$ as the bunch, and with f_1 as remainder;
2. On the second time, we learn CNF "f_2" with $\bigwedge_{i=1}^{n}(\neg x \vee l_i)$ as the basis, with $\bigwedge_{j=1}^{p}(x \vee l'_j)$ as the bunch, and with the **empty** remainder.

In the second case, it is formally not necessary to give f_1 as the remainder: f_1 already appears in the `rup_proofs` field of the bunch. Hence, it is useless to put f_1 in the remainder if we aim to delete it from the current CNF just after.

5.4.3 Formalization of the RAT checker

In order to define and prove the main loop of `unsatProver` with RAT checking, it is convenient to introduce a generic loop, called `loop_until_None`, dedicated to refutation of unreachability properties. This loop – defined in Figure 5.6 – iterates a body of type `S → ??(option S)` until it reaches a `None` value. This body is assumed to preserve an invariant and to never reach `None` under the assumption of this invariant. Hence, if `None` is finally reached, then the invariant was false in the initial state. The `loop_until_None` loop reuses the `loop` oracle of Figure 2.7 and is very similar to the generic WHILE-loop.

At last, we extend our untrusted LRAT parser of Section 5.3.1. As discussed on Example 5.3, "learning" a RAT clause replaces the whole CNF by a new one. Thus, our parser learns RUP clauses until it finds a bunch of RAT clauses. Then, it stops, requiring the CNF to be updated. Afterwards, if the RAT bunch is correct, the certified checker restarts the untrusted parser for the updated CNF. This loop runs until the parser finds an empty RUP clause w.r.t. the current CNF. The untrusted parser,

called `next_RAT` in Figure 5.7, behaves as an iterator over RAT bunches. This iterator is expected to return either the empty clause (left case) or a new RAT bunch to learn (right case). The looping process in `unsatProver` is a simple instance of `loop_until_None`.

Note that—like for our RUP checker—the dictionary of RAT clauses is still only handled by the untrusted LRAT parser. For example, the deletion of clauses is only managed in the untrusted parser. The verified prover simply maintains a list of clauses, globally updated at each learning of a new RAT bunch.

```
Let luni {S} (body: S → ??(option S)) (I: S → Prop) :=
  ∀ s, I s → WHEN (body s) ⇝ s'
             THEN match s' with Some s1 ⇒ I s1 | None ⇒ False end.
Program Definition loop_until_None{S} body (I:S→Prop|luni body I) s0
  : ?? ¬(I s0)
  := loop (A:={s | I s0 → I s})
          (s0, fun s ⇒
               DO s' ↜ mk_annot (body s) ;;
               match s' with
               | Some s1 ⇒ RET (inl (A:={s | I s0 → I s }) s1)
               | None ⇒ RET (inr (B:=¬(I s0)) _)
               end).
```

Figure 5.6: A Generic Loop to Refute Unreachability Properties

```
Axiom next_RAT: ∀ {C}, (rupLCF C) * (list C) → ??(C + RatInput C).
Program Definition unsatProver: ∀ (f:cnf), ?? ¬(∃ m, ⟦f⟧m) :=
  loop_until_None
     (fun f ⇒ (* loop body *)
      DO step ↜ next_RAT (mkInput f) ;;
      match step with
      | inl c ⇒
         assertEmpty (rep c);;
         RET None
      | inr ri ⇒ (* build a new CNF from the RAT bunch *)
         DO f' ↜ learnRat ri;;
         RET (Some f')
       end)
     (fun f ⇒ ∃ m, ⟦f⟧m). (* loop invariant *)
```

Figure 5.7: The RAT prover of {SatAnsCert}

5.5 Performances & Comparison with other works

Our evaluation of {SatAnsCert} is split according to SAT and UNSAT answers. Our SAT benchmark – illustrated in Figure 5.8 – has been established with the CaDiCaL SAT-solver over 120 instances of the SAT competition 2018. Considering the logarithmic scales, the running times of the SAT checker of {SatAnsCert} in Figure 5.8 are negligible w.r.t. those of the solver. And, as expected, the running times of our SAT checker are linear w.r.t the size of the input CNF (being given either in number of clauses or in number of literals).

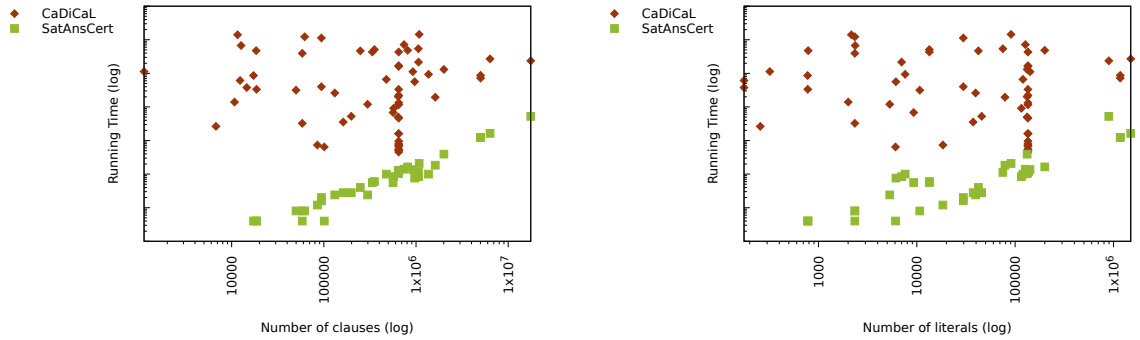


Figure 5.8: Our SAT benchmark based on the CaDiCaL (sc18) SAT-solver

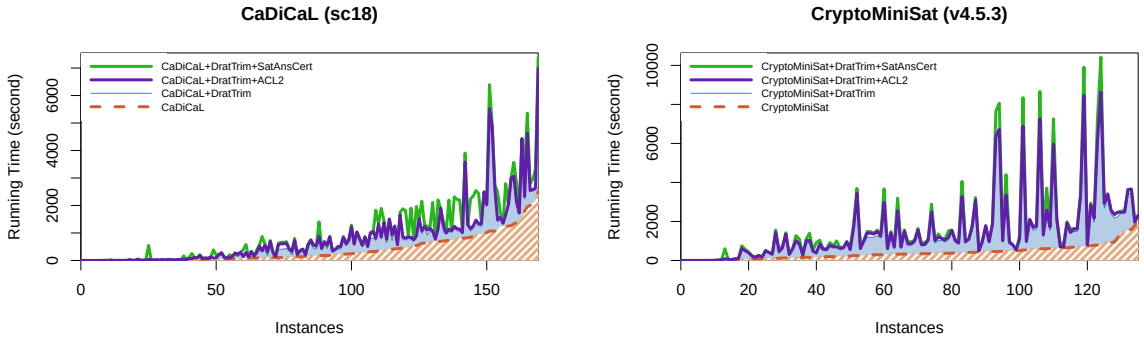


Figure 5.9: Our UNSAT benchmarks

The UNSAT benchmark has been established by using two different solvers: CaDiCaL (sc18) which generates only RUP clauses and CryptoMiniSat (v4.5.3) which produces both RUP and RAT clauses. It is based on more than 170 instances from the SAT competition 2015, 2016 and 2018. Figure 5.9 represents – for each tested instance – the contribution of each tool in the running time, by cumulating their runtimes on upward ordinates. Along the abscissa axis, the instances are ordered by running times of the SAT-solver. By comparing the overhead of the Coq checkers w.r.t DRAT-trim in Figure 5.4 and in Figure 5.9, we see that our LRAT checker is much faster than the Coq/OCaml checker of [Cru+] which has inspired it. We believe that our lightweight design, based on parametric reasoning, has a significant impact on performances here (and it makes the formal proof much more simpler). As also shown in Figure 5.9, our LRAT checker is most often slower than the ACL2/C checker of [Cru+]. We could probably significantly improve the performance of {SatAnsCert}, by encoding literals with native integers instead of Coq positives (aka lists of bits), and by encoding clauses with native persistent arrays instead of radix-trees. These native data-structures were experimentally introduced in Coq by [Arm+10] and had a positive impact on their resolution checker. At the summer 2018, the development time of {SatAnsCert}, they had however still an experimental status in Coq. We did not compare the efficiency of {SatAnsCert} with SMTCoq[11] because its UNSAT prover of Boolean CNF[Kel13] was based on the zChaff SAT-solver[12] which is too old to emit DRUP/DRAT

[11] https://smtcoq.github.io/

[12] http://www.princeton.edu/~chaff/zchaff.html: its last release dates back of 2007. It is now out-performed by more

proofs.

The GRAT toolchain [Lam17b] is an alternative for certified checking of DRAT files. As the DRAT-trim toolchain, it takes a CNF in DIMACS format and a DRAT file in input, generates some intermediate files through an untrusted C++ tool, and gives a certified answer from this intermediate files thanks to an Isabelle/MLton checker. According to [Lam17a], the GRAT toolchain is faster than the DRAT-trim one. Because {SatAnsCert} is itself based on DRAT-trim, we did not find very significant to compare it experimentally to the GRAT toolchain.

In conclusion, {SatAnsCert} is not the most optimized DRAT checker. But the bottleneck of running times in our UNSAT checking is DRAT-trim (the standard checker in SAT competitions). Indeed, on average of the UNSAT benchmark depicted in Figure 5.9, the solver takes 30% of the running time, DRAT-trim takes 50%, and our certified LRAT checker takes the 20% remaining. This demonstrates that {SatAnsCert} reasonably scales up on state-of-the-art SAT-solvers. One of our most noticeable achievement is that {SatAnsCert} results from only a modest effort: we evaluate the whole development at 2 person·months for 1Kloc of Coq (including all proof scripts) and 1Kloc of OCaml files (including `.mll` files). These figures exclude the development of the {Impure} library itself.

recent solvers.

Chapter 6

Toward FVDP without Extraction: Turning a PFS Oracle into a Coq Tactic[†]

If extra security [...] is desired, full proofs are easily generated – only minor changes in the implementation of the abstract type [of theorems] would be required.

Mike Gordon et al. in "*A Metalanguage for Interactive Proof in* LCF" [Gor+78].

A domain-specific language can be implemented by embedding within a general-purpose host language. This embedding may be deep or shallow, depending on whether terms in the language construct syntactic or semantic representations. The deep and shallow styles are closely related, and intimately connected to folds.

Jeremy Gibbons and Nicolas Wu in "*Folding domain-specific languages*" [GW14].

Contents

As detailed in Section 4.2, we designed PFS in order to alleviate our oracles from certificate generation in FVDP through Coq extraction. Yet, certificates are still useful for other applications. For example, certificates could provide a way to reduce the TCB w.r.t. our current approach. We could imagine certifying each run of our OCaml oracles by generating a Coq term representing this run. This term would be dumped in a Coq source file (in Gallina syntax) and checked by the Coq compiler. Coq extraction would no longer be part of the TCB. See also Section 1.2.4.

As another example, Section 6.1 recalls how a Coq tactic—called {VplTactic}—was derived from a {VPL} oracle. This tactic requires an OCaml oracle that produces a Coq AST—i.e. a kind of certificate—typechecked by the Coq kernel. This AST represents a polyhedral computation, itself encoded as a value of a Coq inductive type – similar to the `pexp` type of Section 4.2.2. The tactic then applies a Coq version of the `Front.run` interpreter of Section 4.2.2 to this certificate of type `pexp`.

[†]This chapter summarizes [#BM18] and extends it with an unpublished appendix.

Below, Section 6.2 explains why PFS is also relevant in this case and how certificate are efficiently generated from {VPL} oracles.

6.1 {VplTactic} for Equality Learning in Linear Arithmetic

{VplTactic} is an experimental Coq tactic which simplifies rational inequalities in Coq proofs by doing polyhedral computations [#BM18]. In short, this tactic first reifies the goal into a set of linear inequalities, then either it proves the goal, or it injects as hypotheses a *complete* set of linear equalities that are deduced from the (nonstrict) linear inequalities. Then, many Coq tactics —like `congruence`, `field` or even `auto`—may exploit these new equalities, even if they cannot deduce them from the initial context by themselves. Actually, the idea to use decision procedures which either return "UNSAT" or return a list of learned equalities, is well known in SMT-solving (e.g. Nelson-Oppen approach with Shostak theories [MZ02]). {VplTactic} simply experiments this idea in the context of an interactive theorem prover.

Let us illustrate this feature on the following – almost trivial – Coq goal, where `Qc` is the type of rationals on which our tactic applies.

```
Lemma ex1 (x:Qc) (f:Qc → Qc):  x≤1 → (f x)<(f 1) → x<1.
```

This goal is valid on `Qc` and `Z`, but both `omega` and `lia` fail on the `Z` instance without providing any help to the user. Indeed, since this goal contains an uninterpreted function `f`, it does not fit into the pure linear arithmetic fragment. On the contrary, this goal is proved by two successive calls to the `vpl` tactic.[1] Indeed, the first `vpl` call starts by internally turning the conclusion "`x<1`" of the goal into "`x ≥ 1 → False`" and then rewriting the whole goal into a formula, here pretty-printed as:

$$P[v_1 \mapsto \texttt{x}, v_2 \mapsto (\texttt{f x}), v_3 \mapsto (\texttt{f 1})] \to \texttt{False}$$

where P is the polyhedron $\{C_1 : v_1 \leq 1,\ C_2 : v_2 < v_3,\ C_3 : v_1 \geq 1\}$ with variables v_1, v_2, v_3 in $\mathbb{Q}$.

Then, a {VPL} oracle—detailed below— deduces equality $v_1 = 1$ from P by $\texttt{merge}(C_3, C_1)$ (see Def. 4.4). At last, in the Coq goal, this equality is instantiated as `x=1`, which is then rewritten in `(f x)` (a rewriting that does not fit into linear arithmetic). In summary, the first `vpl` call transforms the initial goal into:

```
x=1
(f 1) < (f 1)
============
False
```

The second call to `vpl` then proves the goal, thanks to the unsatisfiable inequality.

Our oracle learns such equalities from conflicts between strict inequalities. It uses an efficient and simple algorithm which can be viewed as a very specialized optimization for linear arithmetic of Conflict-Driven Clause-Learning (CDCL)—where nonstrict inequalities "$t \geq 0$" are seen as clauses "$t > 0 \vee t = 0$".

[1] Of course, such a goal is also provable in `Z` by SMT-solving tactics: the `verit` tactic of SMTCoq [Arm+11; Eki+17] or the one of Besson et al [BCP11]. However, such SMT-tactics are also "*prove-or-fail*": they do not simplify the goal when they cannot prove it. On the contrary, our tactic may help users in their interactive proofs, by simplifying goals that do not fully fit into the scope of existing SMT-solving procedures.

$$\text{Polyhedron } P_1 \qquad\qquad \text{Polyhedron } P_1^>$$
$$\begin{cases} C_1: & -2x+z-3 \geq 0 \\ C_2: & x-y+z \geq 0 \\ C_3: & -x+5y-7z+6 \geq 0 \\ C_4: & z > 0 \end{cases} \qquad \begin{cases} C_1: & -2x+z-3 > 0 \\ C_2: & x-y+z > 0 \\ C_3: & -x+5y-7z+6 > 0 \\ C_4: & z > 0 \end{cases}$$

Combination $2.C_1 + 5.C_2 + C_3$ gives a proof of $0 > 0$ (UNSAT) when evaluated on $P_1^>$.

This leads to learning two equalities on P_1 $\begin{cases} -4x+2z-6=0 \text{ from } \texttt{merge}(2.C_1, (5.C_2+C_3)) \\ x-5y+7z-6=0 \text{ from } \texttt{merge}((2.C_1+5.C_2), C_3) \end{cases}$

Figure 6.1: Learning Equalities from Conflict on Strict Inequalities

Let us introduce its basic idea[2] on the small example of Figure 6.1. Polyhedron P_1 being given, we first introduce $P_1^>$ by turning each non-strict inequality of P_1 into a strict equality. In Figure 6.1, polyhedron $P_1^>$ is UNSAT as demonstrated by $\Lambda = 2.C_1 + 5.C_2 + C_3$ which is a proof of $0 > 0$. When interpreted in P_1, combination Λ is a proof of $0 \geq 0$. The key idea is that any decomposition of Λ into $\Lambda_1 + \Lambda_2$ splits the proof of $0 \geq 0$ into $t_1 + t_2 \geq 0$ with Λ_1 a proof of $t_1 \geq 0$ (because Λ_1 is a nonnegative combination of P_1 inequalities), Λ_2 a proof of $t_2 \geq 0$ (idem), and with the syntactical equality $t_2 = -t_1$. Hence, for any such a decomposition, we find a proof $\texttt{merge}(\Lambda_1, \Lambda_2)$ in P_1 of $t_1 = 0$ (see examples of Figure 6.1).

In the general case, we learn equalities in this way, as soon as P_1 is SAT and $P_1^>$ is UNSAT. Indeed, the Farkas combination Λ proving the emptiness of $P_1^>$ (see Lemma 4.2) is a contradictory constraint of the form $-q > 0$ with q a nonnegative rational constant. When interpreted on P_1, combination Λ either proves $-q > 0$ or proves $-q \geq 0$. However, because P_1 is SAT, we know that the inequality proved by Λ on P_1 must also be SAT. Thus, Λ is a proof of $0 \geq 0$ on P_1.

In summary, given a polyhedron P—abstracting the hypotheses of the Coq goal—the oracle proceeds in this way:

1. If P is unsatisfiable, then the goal is proved.
2. Otherwise, let us consider a polyhedron $P^>$ derived from P by turning each nonstrict inequality into a strict equality (geometrically, $P^>$ is the *interior* of P). If $P^>$ is unsatisfiable, then there exists a nontrivial Farkas combination Λ proving $0 > 0$ in $P^>$ and $0 \geq 0$ in P (because P is satisfiable). Assuming that Λ is given by a sum of $n+1$ inequalities, we deduce n equalities from Λ. Then, we replace in P these $n+1$ inequalities by these n equalities (and also simplifies P according to these equalities). At last, we iterate this step until $P^>$ becomes satisfiable, and apply case 3 below.
3. When $P^>$ is satisfiable, there is no new equality to learn. The oracle still simplifies P by removing all redundant inequalities.

As explained above, in this algorithm, the Farkas combinations proving the new equalities are obtained by very simple rewritings of the Farkas combination Λ proving $0 > 0$ on $P^>$. Considering that Λ is a linear combination of $n+1$ inequalities, then each of the learned equalities is proved by replacing in Λ one of the n additions (operator "+") by a conjunction (operator `merge`). In other words, the proofs of the discovered equalities are learned from the proof of a conflict. Certificate generation and clause-learning are closely related, as already noted in Section 5.2.1. See [#BM18, Sect. 5] for a more rigorous presentation of this algorithm.

[2]This simple presentation assumes that constraints of polyhedra are given in the form of "$t \bowtie 0$" with $\bowtie \in \{\geq, >\}$. In the actual algorithm, we work on a echelon system where explicit equalities are aside. See [#BM18] for details.

In summary, the {VPL} oracle involved in the {VplTactic} performs the simplification of a polyhedron P into a polyhedron P' such that $P \Rightarrow P'$ and P' is *reduced* [#BM18]. This simplification is itself implemented as a PFS oracle, using the extended Farkas factory of Section 4.3.1. However, Coq expects that tactics produce proof terms. To this purpose, {VplTactic} internally defines an AST type of certificates, extending type `pexp` of Section 4.2.2. In other words, {VplTactic} introduces a *deep embedding* of this language of certificates. Section 6.2 now explains why PFS is still relevant for programming oracles embedded by such a Coq tactic and how certificates are efficiently generated from PFS oracles.

6.2 Generating Compact Certificates from PFS Oracles

Polymorphic factories provide an abstract layer that simplifies the implementation of certificate generating oracles. Indeed, if we consider a given language of certificates as a DSL (Domain Specific Language), then the polymorphic factory corresponds to a generic *shallow embedding* of this language in oracles.[3] Such a shallow embedding alleviates the development of oracles, because some construct and transformations on the DSL can be directly emulated with ML constructs. For example, Section 4.2 illustrates the interest of emulating a `Bind` operator (Sect. 4.2.2) by a ML "**`let in`**"; Section 4.3 illustrates how explicit variable substitutions over DSL terms are just avoided, because variables of the DSL are directly ML variables (hence substitutions of variables are simply emulated by applying ML functions). Moreover, the ability to instantiate the factory helps in debugging oracles. Finally, the code that generates certificates (i.e. the AST of the corresponding *deep embedding*) is easily factorized for a family of oracles, as illustrated in Section 6.2.1.

Below, by defining a well chosen factory, we produce a compact AST without slowing too much its generation. This factory actually produces a DAG, from which the final AST is extracted after a dependency analysis. For example, intermediate results that are actually not needed for the AST are discarded. Similarly, when an intermediate computation is used at least twice, we define a binder that stores this result into an intermediate variable. These two optimizations, explained in Section 6.2.3, avoid useless or redundant computations in the AST interpreter. Another optimization is performed on the DAG: `top` nodes are eliminated, and multiplications by constants are factorized. Section 6.2.2 gives the factory that produces the DAG, and how this last optimization is applied on the fly.

6.2.1 Factorizing the AST Generation from PFS Oracles

The DAG data structure provides the interface below, which helps to wrap PFS oracles of the {VPL}. Type `dsctr` is the type of nodes in the DAG. Constant `dag_factory` provides a factory instance for our PFS oracles. Function `import` converts an input polyhedron into an input suitable for oracles. Finally, function `export` converts the output of oracles into an AST of type `pexp`.

```
type dcstr
val dag_factory: dcstr Back.fLCF
val import: BackCstr.t list -> (BackCstr.t * dcstr) list
val export:  ('a * dcstr) list -> pexp
```

From this interface, wrapping a given PFS oracle into an AST producing oracle is straightforward. For example, we define below `ast_proj` which wraps the `Back.proj` PFS oracle of Section 4.2.4.

[3] Actually, extending Gibbons and Wu's explanations [GW14], the polymorphic factory could be viewed as the parameter of a given polymorphic recursion operator—also usually called a "fold" operator—over the AST of the corresponding deep embedding.

```
let ast_proj (p: BackCstr.t list) (x: Var.t): pexp =
  export (Back.proj dag_factory (import p) x)
```

Below, Section 6.2.2 defines `dag_factory` and `import` that makes the PFS oracle builds the DAG. Section 6.2.3 describes the analysis of this DAG in `export` to produce a compact AST.

6.2.2 A Factory Producing a DAG

For the sake of simplicity, we illustrate the generation of the DAG on the following sub-factory of the one of Section 4.3.1.

```
type 'c fLCF = { top: 'c; add: 'c -> 'c -> 'c; mul: Rat.t -> 'c -> 'c }
```

During the DAG generation, we eliminate the neutral element `top` that introduces useless nodes. We also factorize multiplications by rational constants. These propagations are directly achieved by the operations of `dag_factory`.

The type `dcstr` of nodes in the DAG is implemented as shown on the right. This is a record type with a field `def` containing the "operation" at this node. An operation of type `op` corresponds either to an input constraint (constructor `_Ident`) or to an operation on constraints. Operations `_Add` and `_Mul` refer to nodes of type `dcstr`, and such a node can be shared between several operations by pointer sharing. Mutable fields of `dcstr`, like `id` and `nbusers`, are only used during function `export`. They represent auxiliary data on the node, which are computed by the dependency analysis and useful to generate the final AST.

```
type dcstr = {
  def: op;
  mutable id: int;
  mutable nbusers: int;
  (* other omitted fields *)
} and op =
  | Ident_
  | Top
  | Add_ of dcstr * dcstr
  | Mul_ of Rat.t * dcstr
```

We call a node `dc1` a *direct ancestor* of a node `dc2` iff `dc2` appears in `dc1.def` (i.e. as arguments of `Add_` or `Mul_`). It corresponds to the fact that the computation represented by `dc1` *depends on* the result of the computation represented by `dc2`. Here, `dc2` is a reference that may have several direct ancestors but, by construction, it can not be a direct or indirect ancestor of itself.

Most new nodes of the DAG are generated through a call to `(make_dcstr d)` where `d` is a value of type `op`. This call initializes field `def` with value `d` and other fields with default values (these latter being only used in `export`). The only exception is on `Ident_` nodes that are created with a positive field `id` giving their name in the final AST.

```
let make_dcstr ?id:(i=0) d : dcstr = { def=d; id=i; nbusers=0; (* ... *) }
```

Let us now detail the implementation of `import` and `dag_factory`. On a given polyhedron `p`, function `import` associates a new `Ident_` node to each constraint `c` of `p`. The name of each of these nodes – given by its field `id` – corresponds to the position of `c` in the list `p`.

```
let import p = List.mapi (fun i c -> (c, make_dcstr ~id:(i+1) Ident_)) p
```

In `dag_factory`, functions `smart_mul` and `smart_add` are *smart constructors* of nodes which eliminate `Top` nodes and factorize `Mul_` nodes as much as possible.

```
let dag_factory = {top = make_dcstr Top; add = smart_add; mul = smart_mul}
```

$$n \cdot \top \to \top \qquad 1 \cdot c \to c \qquad n_1 \cdot (n_2 \cdot c) \to (n_1 \times n_2) \cdot c$$

$$\top + c \to c \qquad c + \top \to c \qquad (n_1 \cdot c_1) + (n_2 \cdot c_2) \to \begin{cases} n_1 \cdot \left(c_1 + \frac{n_2}{n_1} \cdot c_2\right) & \text{if } n_1 > 0 \\ n_2 \cdot \left(\frac{n_1}{n_2} \cdot c_1 + c_2\right) & \text{if } n_1 < 0 \end{cases}$$

Figure 6.2: Elimination of `Top` Nodes and Factorization of `Mul_` Nodes in the DAG

This process corresponds to applying the rewriting rules of Figure 6.2, where $\top$, + and · represent a node where the field `def` is respectively `Top`, `Add_` and `Mul_` and where c, c_1 and c_2 are some other existing nodes. Since these smart constructors assume that their node in inputs are *already rewritten*, they only perform $O(1)$ rewriting steps at each call. Moreover, `(smart_mul n c)` assumes that scalar `n` is not zero and that if `n` is negative then `c` is an equality. These two last assumptions are of course valid on our PFS oracles, and they are preserved by the rewriting rules of Figure 6.2.

For instance, on a witness "$n_1 \cdot c_1 + n_2 \cdot \left(\top + \frac{n_1}{n_2} \cdot c_2\right)$" generated from a PFS oracle (where $n_1 > 0$), the factory builds a node corresponding to "$n_1 \cdot (c_1 + c_2)$". Let us remark that some useless nodes, such as "$\frac{n_1}{n_2} \cdot c_2$", are generated in the DAG during this process. But they do not pollute the final AST, thanks to the dependency analysis of the next section.

6.2.3 Producing the AST

We aim here to produce certificates like examples given in Figure 4.2 at page 74, where derived constraints used in at least two Farkas combinations (of type `fexp`) are named by a `Bind` instead of having their combination duplicated. In the DAG, each node leading to the introduction of a `Bind` in the final AST is marked by the dependency analysis with a unique positive integer i in their field `id`. Hence, it is converted as "`Ident` i" in Farkas combinations of the final AST.

Let us detail the implementation of `export`, that builds the AST from the oracle result `res`. The function extracts `dsctr` nodes and discards `Top` nodes, which are ignored without loss of information. This gives a value `roots:(dsctr list)` corresponding to the set of roots in input of the dependency analysis – named `compute_defs`. This function analyzes descendants of the roots in the DAG, and returns the list `defs` of the non-`_Ident` nodes that have at least two direct ancestors (among descendants of the roots). In list `defs`, nodes are sorted following a topological sort with ancestors first and are named with unique positive integers (in field `id`) above the maximum name of reachable `_Ident` nodes. At last, given lists `defs` and `roots`, function `mk_pexp` produces the final AST.

```
let export res =
  let roots = List.fold_left
    (fun acc (_,dc) -> if dc.def<>Top then (dc::acc) else acc) [] res in
  (* compute_defs: dcstr list -> dcstr list *)
  let defs = compute_defs roots in
  (* mk_pexp: dcstr list -> dcstr list -> pexp *)
  mk_pexp defs roots
```

The code of `mk_pexp` is given below. Each node of `roots` leads to a Farkas combination – built by function `mk_fexp` – under the `Return` node (i.e. it represents a constraint in the final polyhedron). Here, nodes with a null `id` field are used exactly once in the AST. Each node `dc` of `defs` induces a `Bind` node where its Farkas combination – built by function `mk_def` – is associated to the positive name `dc.id`.

```
(* val mk_fexp: dcstr -> fexp *)
let rec mk_fexp dc =
  if dc.id > 0 then (Ident dc.id) else (mk_def dc)
and mk_def dc =
  match dc.def with
  | Add_ (dc1, dc2)  -> Add (mk_fexp dc1, mk_fexp dc2)
  | Mul_ (n, dc) ->  Mul (n, mk_fexp dc)
  | _ -> assert false

let mk_pexp defs roots =
  let lr = Return (List.map mk_fexp roots) in
  List.fold_left (fun rem dc -> Bind (dc.id, mk_def dc, rem)) lr defs
```

In summary, our `export` function builds an compact AST using a named representation in binders, and where unbound names represent input constraints. If necessary, it is easy to convert this representation for a de Bruijn representation of binders and where input constraints are introduced by a dedicated binder.

Chapter 7

FVDP of Impure Abstract Interpretations by Stepwise Refinement[†]

We have shown that the classical semantics of programs [...] can be derived from one another [...]. Our presentation uses abstraction which proceeds by omitting some aspects of program execution but the inverse operation of semantic refinement (traditionally called concretization) is equally important. This suggests considering hierarchies of semantics which can describe program properties, that is program executions, at various levels of abstraction or refinement in a uniform framework.

Patrick Cousot, in "*Constructive Design of a Hierarchy of Semantics ...*" [Cou02].

Stepwise refinement was originally proposed by Dijkstra [Dij68] as a constructive approach *to program proving. According to this view, if each refinement step is very carefully carried out, so it can be seen to preserve the correctness of the previous version of the program, then the final program must be correct* by construction.

Ralph-Johan Back, in "*Correctness Preserving Program Refinements*" [Bac80].

Contents

[†]This chapter has been published in [#BM19], which extends [#BM15].

This chapter deals with the modular development of formally verified static analyzers in the Coq proof assistant. We focus on the implementation in the {VPL} of a linearization procedure to handle polynomial guards. Based on ring rewriting strategies and interval arithmetic, this procedure partitions the variable space to infer precise affine terms which over-approximate polynomials.

In order to help formal development, we propose a proof framework, embedded in Coq, that implements a refinement calculus. It is dedicated to certifying the components of the analyzer – like our linearization procedure – for which the correctness does not depend on the implementation of the underlying certified abstract domain. Like standard refinement calculi, it introduces data-refinement diagrams. These diagrams relate "*abstract states*" computed by the analyzer to "*concrete states*" of the input program. However, our notions of "*specification*" and "*implementation*" are exchanged w.r.t. standard uses: the "*specification*" (computing on "*concrete states*") refines the "*implementation*" (computing on "*abstract states*"), because here, we want to prove the correctness of some computations in the abstract domain w.r.t the concrete semantics.[1]

Our stepwise refinements hide several low-level aspects of the computations on abstract domains. In particular, they ignore that the latter may use hints from external untrusted imperative oracles (e.g. a linear programming solver). Moreover, refinement proofs are naturally simplified thanks to computations of weakest preconditions. Using our refinement calculus, we simply implement the partitioning strategies of our linearization procedure with a continuation-passing style, thus avoiding an explicit datatype of partitions. This illustrates that our framework is convenient to prove the correctness of such higher-order imperative computations on abstract domains.

7.1 Introduction

We now introduce these two contributions: first, a certified linearization procedure for the {VPL}; second, a refinement calculus to help in mechanizing this proof in Coq. We detail below the context and features of these two contributions.

7.1.1 A Certified Linearization for the Abstract Domain of Polyhedra

As introduced in Section 4.1, the {VPL} provides a formally verified *abstract domain of convex polyhedra*, where invariants are conjunctions of affine constraints written $\sum_i a_i x_i \leq b$ where $a_i, b \in \mathbb{Q}$ are scalar values and x_i are integer program variables. This domain is able to capture relations between program variables (e.g. $x+2 \leq y+x-2z$). However, it cannot deal directly with non-linear invariants, such as $x^2 - y^2 \leq x \times y$. This is why linearization techniques are necessary to analyze programs with non-linear arithmetic.

[1] Indeed, we consider that in a static analyzer, like in a compiler, the "*source program*" is (a part of) the specification against which we prove the transformations. Moreover, w.r.t. the terminology of abstract interpretation, we find clearer to keep the notion of "abstract" and "concrete" unchanged and to keep saying that "the concrete refines the abstract". This leads us to say that, in abstract interpretation, "the specification refines the implementation".

Our certified linearization procedure is based on intervalization [Min06]. It consists in replacing some variables of nonlinear products by intervals of constants. For instance, Example 7.1 replaces variable x by interval $[0, 10]$ in product "$x.(y - z)$". The interval is then eliminated by analyzing the sign of $y - z$, leading to affine constraints usable by the polyhedra domain.

Example 7.1 (Intervalization using a sign-analysis)**.** In any state where $x \in [0, 10]$, concrete assignment "$r := x.(y - z) + 10.z$" is approximated by the affine program below. Here operator $:\in$ performs a nondeterministic assignment.

```
if y − z ≥ 0 then r :∈ [10.z, 10.y] else r :∈ [10.y, 10.z]
```

In other words, r is updated to any value of $[\min(10.y, 10.z), \max(10.y, 10.z)]$.

Let us clearly delimit the scope of our work. Our linearization procedure is part of the {VPL}, which provides a certified polyhedra domain to Verasco [Jou+15; Lap15; Jou16], a certified abstract interpreter for CompCert C [Ler09b]. Verasco is a static analyzer that checks that C programs have no undefined behavior. Hence, our refinement calculus focuses on abstract interpretations that overapproximate sets of reachable states and that reject reachable error states. For example, our refinement calculus is not sufficient to prove (alone) the correctness of an abstract interpretation bounding execution times of programs.[2] Second, the {VPL} abstract domain in Verasco is limited to integer variables and rational constants. It could also support rational variables. But supporting floating-point operators would be a non-trivial extension.

As detailed in Chapter 4, the {VPL} relies on {Impure} untrusted oracles. Built on a similar design, our linearization procedure invokes an untrusted oracle[3] that selects strategies for linearizing an arithmetic expression and produces a certificate that is checked by the certified part of the procedure. It leads to a correct-by-construction over-approximation of the expression. It is convenient to see such strategies as program transformations, because their correctness is independent from the implementation of the underlying abstract domain and is naturally expressed using concrete semantics of programs. Indeed, a linearization is correct if, in the current context of the analysis, any postcondition satisfied by the output program is also satisfied by the input one (see Example 7.1). In such a case, we say that the input program *refines* the output one. This chapter aims to explain how refinement helps develop certified procedures on abstract domains, such as our linearization algorithm.

7.1.2 Refinement to Certify Computations on Abstract Domains

Program refinement [BvW98; Mor94] consists in decomposing proofs of complex programs by stepwise applications of correctness-preserving transformations. We provide a framework in Coq to apply this methodology for certifying the correctness of computations combining operators of an existing abstract domain: our goal is to compositionally build correct-by-construction abstract computations, by reasoning only on the concrete semantics of programs.

Typically, an affine program – like in Example 7.1 – is both interpreted in our abstract and concrete semantics. We thus reduce the proof that the abstract interpretation of this affine program computes

[2] Of course, our refinement calculus may help in statically verifying that a program *enriched with assertions enforcing a bounded execution* will satisfy these assertions. But, it will not help to prove that these assertions are sufficient for enforcing the execution to be bounded.

[3] There are several kinds of oracles in the {VPL}: those based on Farkas polymorphic factory for basic polyhedra computations of Section 4.3.1; the linearization strategy in the linearization procedure; etc. In the Coq code, each of these oracles is declared as a "nondeterministic" function in parameter of the code with an ancester of the {Impure} library presented in Chapter 2.

a correct overapproximation of the input program to the proof that its concrete interpretation refines the input program. This proof may be itself composed of several stepwise refinements in the concrete semantics. Indeed, the development of our linearization procedure extends concrete semantics with *affine interval arithmetic* [Min06] (i.e. affine arithmetic where constants are replaced by intervals of constants). In this approach, refinement of Example 7.1 is decomposed into two refinement steps given in Example 7.2. Here, assumption $x \in [0, 10]$ is reflected in the input program syntax thanks to an `assume` command (formally defined in Section 7.2.1).

Example 7.2 (Refinement steps)**.** The affine program

$\texttt{if}\ \ y - z \geq 0\ \ \texttt{then}\ \ r :\in [0 + 10.z, 10.(y - z) + 10.z]\ \ \texttt{else}\ \ r :\in [10.(y - z) + 10.z, 0 + 10.z]$

is refined by

$r :\in [0, 10].(y - z) + 10.z$

itself refined by

$\texttt{assume}\ \ x \in [0, 10]\texttt{;}$ $r := x.(y - z) + 10.z$

On Example 7.2, the first refinement step reduces to two properties of interval multiplication

$$\begin{aligned} y - z \geq 0 &\ \Rightarrow\ [0, 10].(y - z) = [0, 10.(y - z)] \\ y - z < 0 &\ \Rightarrow\ [0, 10].(y - z) = [10.(y - z), 0] \end{aligned}$$

The program in the middle just aims at simplifying proofs. Indeed, the second refinement step reduces to

$$x \in [0, 10]\ \Rightarrow\ x.(y - z) + 10.z \in [0, 10].(y - z) + 10.z$$

This property trivially results from composition properties of interval arithmetic operators. Thus, this whole proof completely ignores that our abstract interpretation of the first program involves imperative computations using a given representation of polyhedra.

7.1.3 Overview of our Refinement Calculus

Our framework defines a Guarded Command Language (GCL) called $^{\dagger}\mathbb{K}$ that contains the basic operators of the abstract domain. A computation $^{\dagger}K$ in $^{\dagger}\mathbb{K}$ comes with two types of semantics: an abstract and a concrete one. Concrete semantics of $^{\dagger}K$ is a transformation on *memory states*. Abstract semantics of $^{\dagger}K$ is a transformation on *abstract states*, i.e. on values of the abstract domain. A $^{\dagger}\mathbb{K}$ computation also embeds a proof that abstract semantics is correct w.r.t. concrete one: each $^{\dagger}\mathbb{K}$ operator thus preserves correctness by definition. Moreover, an OCaml function is extracted from abstract semantics, which is certified to be correct w.r.t. concrete semantics. Hence, concrete semantics of $^{\dagger}K$ acts as a *specification* which is *implemented* by its abstract semantics. In the following, a transformation on abstract (resp. memory) states is called an abstract (resp. concrete) computation.

Taking a piece of code as input, our linearization procedure outputs a $^{\dagger}\mathbb{K}$ computation. Its correctness is ensured by proving that concrete semantics of its input refines concrete semantics of its output. This means that the output does not forget any behavior of the input. Our procedure being developed in a modular way from small intermediate functions, its proof reduces itself to small refinement steps.[4] Each of these refinement steps is only proved by reasoning on the concrete semantics.

[4]Thus, we do not use our refinement calculus in a *decompositional* (i.e. "top-down") approach, that builds an implementation by stepwise derivation from a specification. On the contrary, we use our refinement calculus in a *compositional* (i.e. "bottom-up") approach, that builds larger "bricks" from smaller "bricks".

Our framework provides a tactic simplifying such refinement proofs by computational reflection of weakest-preconditions in the concrete semantics[5]. The correctness of abstract semantics w.r.t. concrete semantics is ensured by construction of $^{\dagger}\mathbb{K}$ operators.

Our framework supports *impure* abstract computations, i.e. abstract computations that invoke imperative oracles giving them hints to build their certified results. It also allows to reason conveniently about higher-order abstract computations. In particular, our linearization procedure uses a Continuation-Passing Style (CPS) [Rey93] in order to partition its analyzes according to the sign of affine sub-expressions. For instance in Example 7.2, the approximation of the non-linear assignment depends on the sign of $y - z$. In our procedure, CPS is a higher-order programming style that avoids introducing an explicit datatype handling partitions: this simplifies both the implementation and its proof. This also illustrates the expressive power of our framework, since a simple Hoare logic does not suffice to reason about such higher-order imperative programs.

Our refinement calculus could have applications beyond the correctness of linearization strategies: it could be applied for any part of the analyzer that combines computations of existing abstract domains. In particular, the top-level interpreter of the analyzer could also be proved correct in this way. Indeed, the interpreter invokes operations on abstract domains in order to over-approximate any execution of the program, but its correctness does not depend on abstract domains implementations (as soon as these implementations are themselves correct). We illustrate this claim on a toy analyzer, also implemented in Coq. Let us explain this contribution w.r.t. the certification of the top-level interpreter of Verasco developed by Jacques-Henri Jourdan [Jou16].

The interpreter of Verasco analyzes `C♯minor` [Ler09b] – an intermediate structured language of CompCert frontend – w.r.t. its small-step semantics. Actually, this small-step semantics (from CompCert) introduces many low-level details that are tedious to deal with in the proof of the analyzer.[6] Thus, Jourdan has introduced a higher-level semantics of `C♯minor` in order to simplify his proof. This semantics is a Hoare logic because such a logic is better suited to *structured* languages than usual collecting semantics which are dedicated to *Control Flow Graph* representations [Jou16]. Hence, Jourdan's proof is realized in a framework combining a Hoare logic as concrete semantics, with a theory of abstract domains. But Jourdan's framework assumes that operators of abstract domains are pure functions. Actually, this is not the case of {VPL} operators.[7]

Our refinement calculus sketches an alternative to Jourdan's framework in order to support *impure* operators in abstract domains. Our toy analyzer illustrates how the refinement calculus helps mechanize the correctness proof of the interpreter. Moreover, it also illustrates that alarm handling of Verasco is very easy to support in our framework. However, our interpreter does not support many other features of Verasco interpreter: control-flow statements such as "`break`" and "`continue`", the inference mechanism of loop invariants[8], communication between several abstract domains, etc.

[5]These weakest-preconditions on the concrete semantics are defined in Sect. 7.4.2. They should not be confused with the `wlp_simplify` tactic of Sect. 2.2.1. As discussed in Sect. 7.4.3, in the context of this chapter, `wlp_simplify` is only used to prove the correctness of the basic operators provided by $^{\dagger}\mathbb{K}$.

[6]Typically, `C♯minor` small-step semantics distinguishes infinite loops depending on whether they produce observational events or not. But, by definition, an infinite loop cannot have runtime errors. Hence, all infinite loops are equivalent for Verasco analyzer. Even better, the analyzer can safely prune any control-flow branch where they appear, exactly like unreachable code.

[7]Thus, the embedding of {VPL} in Verasco coerces its imperative operators into pure ones. Logically, this coercion remains to assume that {VPL} oracles are observationally pure. This is potentially wrong, because of potential bugs in these untrusted oracles. See Section 2.1.

[8]Our toy analyzer does not infer loop invariants but requires them from the user. It does not seem too hard to extend our analyzer with inference of loop invariants since the {VPL} provides a standard (untrusted) widening operator. But, this feature is quite orthogonal to the certification of the analyzer itself. For example, Laporte [Lap15] shows how to program such an untrusted oracle, in order to produce invariants checked by the certified analyzer.

7.1.4 Comparison with Related Works

The mathematics involved in our refinement calculus, relating operational semantics to the lattice structure of monotone predicate transformers, are well-known in abstract interpretation theory [Cou02]. In parallel to our work, the idea to use a refinement calculus in formal proofs of abstract interpreters was also proposed in [Spi13]. Therefore, our contribution is more practical than theoretical. On the theoretical side, we propose a refinement calculus dedicated to the certification of *impure* abstract computations (w.r.t. big-step operational semantics). On the practical side, we show how to get a concise implementation of such a refinement in Coq and how it helps on a realistic case study: a linearization technique inspired from [Min06] within the abstract interpreter Verasco.

There are alternatives to our approach for computing polyhedral approximations of semi-algebraic sets. Let us briefly compare them with intervalization. A linearization procedure based on Handelman representation of polynomials [Han88] has also been implemented in the {VPL} [Mar+16]. It is more precise than intervalization, but at a high cost: it requires solving costly parametric linear problems. Albeit powerful, Handelman's linearization does not scale properly to large polynomials and polyhedra, this is why we need a cheaper algorithm such as intervalization. Another precise approach consists in converting the polynomial into Bernstein's basis and extract the generators of the resulting polyhedron from the polynomial's coefficients [Far12]. Like Handelman's linearization, it offers a tunable precision: either by partitioning the variable space or by elevating the degree of the Bernstein's basis considered. However, in order to ease the certification, the {VPL} uses a constraint-only representation of polyhedra. Using Bernstein's linearization would thus involves costly conversions from constraints into generators, and backwards [Mar17, Chap 4].

There are also linearizations dedicated to other target domains. For instance, a decision procedure for arithmetic that uses affine forms instead of polyhedra has been proven in PVS [MMS15]. In their approach, affine approximations of polynomials are combined with partitioning through a branch-and-bound algorithm. The expressiveness of affine forms is strictly between intervals and polyhedra, but our linearization procedure would probably be greatly improved by incorporating their techniques. Another technique dynamically tunes the trade-off between efficiency and precision thanks to an abstraction-refinement loop within a SMT-solver [Cim+18].

7.1.5 Overview of the Chapter

Our refinement calculus is implemented in only 350 lines of Coq (proof scripts included), by a shallow-embedding of our GCL $^{\dagger}\mathbb{K}$ which combines computational reflection of weakest-preconditions [Dij75] with monads [Wad95]. However, it can be understood in a much simpler setting using binary relations instead of monads and weakest-preconditions, and classical set theory instead of Coq.

Section 7.2 introduces our refinement calculus in this simplified setting, where computations are represented as binary relations. Section 7.3 presents our certified linearization procedure and how its proof benefits from our refinement calculus. Section 7.4 explains how we mechanize this refinement calculus in Coq by using smart encodings of binary relations introduced in Section 7.2.

This chapter is intended to be self-contained. Assuming that the reader is familiar with higher-order logic, big-step semantics and Hoare logic, it attempts to introduce as simply as possible all other notions: refinement, abstract interpretation, convex polyhedra, monads and weakest-preconditions. Our Coq sources are available as a standalone library:

- either at http://www-verimag.imag.fr/~boulme/vpl201503 (first release)
- or at http://github.com/VERIMAG-Polyhedra/VPL (current release)

The version integrated with VERASCO 1.3 is available at
http://compcert.inria.fr/verasco/release/verasco-1.3.tgz

7.2 A Refinement Calculus for Abstract Interpretation

We consider an analyzer correct if and only if it rejects all programs that may lead to an *error state*. Due to lack of precision, it may also reject safe programs. Section 7.2.1 defines the notion of error state and semantics of concrete computations, which combines big-steps operational semantics with Hoare Logic. After introducing the notion of abstract computation and its correctness w.r.t. a concrete computation, Section 7.2.2 presents our refinement calculus. Section 7.2.3 shows how to apply refinement to the certification of higher-order abstract computations.

Notations on Relations. Although our formalization is in the intuitionistic type theory of Coq without axioms, the chapter abusively uses more common notations of classical set theory. In particular, we identify the type $A \to \texttt{Prop}$ of predicates on A with the set $\mathcal{P}(A)$. Hence, we define the set of binary relations on $A \times B$ by $\mathcal{R}(A,B) \triangleq \mathcal{P}(A \times B)$. Given R of $\mathcal{R}(A,B)$, we note $x \overset{R}{\to} y$ instead of $(x,y) \in R$. We use operators on $\mathcal{R}(A,A)$ inspired from regular expressions: ε is *the identity relation* on A, relation $R_1 \cdot R_2$ is "R_2 *composed with* R_1" (i.e. $x \overset{R_1 \cdot R_2}{\longrightarrow} z \triangleq \exists y, x \overset{R_1}{\to} y \wedge y \overset{R_2}{\to} z$) and R^* is the reflexive and transitive closure of R. Through all the chapter, $A \to B$ is a type of *total* functions.

7.2.1 Stepwise Refinement of Concrete Computations

Given a domain D representing the type of memory states, we add a distinguished element $\lightning$ to D in order to represent the error state: we define $D_\lightning \triangleq D \uplus \{\lightning\}$.

Concrete Computations With Runtime Errors. We define the set of computations on memory states, called here *concrete computations*, as $\mathbb{K} \triangleq \mathcal{R}(D, D_\lightning)$. Hence, an element K of $\mathbb{K}$ corresponds to a (possibly) nondeterministic or non-terminating computation from an *input state* of type D to an *output state* of type $D_\lightning$. Typically, the empty relation represents a computation that loops infinitely for any input. It also represents unreachable code i.e. dead code (as explained in Footnote 6).

In the following, an input $d \in D$ is said to be *erroneous for a concrete computation* K if and only if $d \overset{K}{\to} \lightning$. Informally speaking, we consider that an abstract computation is correct w.r.t. a concrete computation K at two conditions: first, it overapproximates the set of erroneous inputs of K as a set E; second, for each input of $D \backslash E$, it overapproximates the set of its related outputs through K. Section 7.2.2 formalizes this notion of abstract computation. Before that, we introduce structures on concrete computations in order to use them as *specifications* of abstract computations.

Refinement Pre-order. Given K_1 and K_2 in $\mathbb{K}$, we say that "K_1 *refines* K_2" (written $K_1 \sqsubseteq K_2$) if, informally, each abstract computation correct for K_2 is also correct for K_1. Let us now formalize this refinement relation.

First, we introduce $\downarrow K$ the normalization of K that returns any output for erroneous inputs. It is defined by $d \overset{\downarrow K}{\longrightarrow} d' \triangleq (d \overset{K}{\to} d' \vee d \overset{K}{\to} \lightning)$. Informally speaking, "adding" some outputs to K on its erroneous inputs does not change the set of abstract computations that are correct w.r.t. K. In other words, an abstract computation is correct for K *if and only if* it is correct w.r.t. $\downarrow K$. Moreover, $\downarrow K$ is the *maximal relation* which is equivalent to K w.r.t. (correct) abstract interpretation.

Then, normalization enables us to define refinement from inclusion. Formally, we define $K_1 \sqsubseteq K_2$ as $K_1 \subseteq {\downarrow}K_2$ (or equivalently, ${\downarrow}K_1 \subseteq {\downarrow}K_2$). Relation $\sqsubseteq$ is called *refinement* and is a pre-order on $\mathbb{K}$. The equivalence relation $\equiv$ associated with this pre-order is given by $K_1 \equiv K_2$ iff ${\downarrow}K_1 = {\downarrow}K_2$.

Hoare Specifications. Hoare logic is a standard and convenient framework to reason about imperative programs. Let us explain how computations in $\mathbb{K}$ are equivalent to specifications of Hoare logic. A computation K corresponds to a Hoare specification $(\mathfrak{p}_K, \mathfrak{q}_K)$ of $\mathcal{P}(D) \times \mathcal{R}(D, D)$, where $\mathfrak{p}_K$ is a precondition ensuring the absence of error, and $\mathfrak{q}_K$ a postcondition relating the input state to a non-error output state[9]. They are defined by $\mathfrak{p}_K \triangleq D \setminus \{d \mid d \xrightarrow{K} \lightning\}$ and $\mathfrak{q}_K \triangleq K \cap (D \times D)$. Conversely, any Hoare specification (P, Q) corresponds to a computation $\vdash P; Q$ – defined below – such that $K \equiv \vdash \mathfrak{p}_K; \mathfrak{q}_K$. Moreover, the refinement pre-order $K_1 \sqsubseteq K_2$ is equivalent to the usual refinement of specifications in Hoare logic, which is $\mathfrak{p}_{K_2} \subseteq \mathfrak{p}_{K_1} \ \wedge \ \mathfrak{q}_{K_1} \cap (\mathfrak{p}_{K_2} \times D) \subseteq \mathfrak{q}_{K_2}$.

Algebra of Guarded Commands. Initially proposed by [Dij75], guarded commands are also equivalent to Hoare specifications, but with an algebraic style, more suited for the methodology of stepwise refinement [BvW98]. Inspired by this methodology, we equip $\mathbb{K}$ with an algebra of guarded commands.[10] It combines a *complete* lattice structure with operators inspired from regular expressions. Here, we present this algebra in our simplified setting, where $\mathbb{K}$ is defined as $\mathcal{R}(D, D_\lightning)$. Our Coq implementation, described in Section 7.4.2, has a different representation of $\mathbb{K}$ in order to mechanize refinement proofs.

First, the complete lattice structure of $\mathbb{K}$ (for pre-order $\sqsubseteq$) is given by operator $\sqcap$ defined as "$\cap$ after normalization" (i.e. $\bigsqcap_i K_i \triangleq \bigcap_i {\downarrow}K_i$) and by operator $\sqcup$ defined as $\cup$. In our context, $\sqcup$ represents alternatives that may non-deterministically happen at runtime: the analyzer must consider that each of them may happen. Assuming that the concrete execution may run K_1 or may run K_2, the analyzer must find an approximation of $K_1 \sqcup K_2$ (which satisfies $K_1 \sqsubseteq K_1 \sqcup K_2$ and $K_2 \sqsubseteq K_1 \sqcup K_2$). Symmetrically, $\sqcap$ represents some choice left to the analyzer. Given a concrete computation K, we may find two distincts approximations K_1 and K_2 such that $K \sqsubseteq K_1$ and $K \sqsubseteq K_2$. In an intermediate stepwise refinement, we may specify this opportunity for the analyzer as $K_1 \sqcap K_2$ (which satisfies $K \sqsubseteq K_1 \sqcap K_2$): this means that the analyzer has the choice to approximate K as K_1 or to approximate it as K_2, or even as the intersection of these approximations (which could give a more precise result).

The empty relation $\emptyset$ is the bottom element and is written $\bot$. The relation $D \bowtie \{\lightning\}$ is the top element. Given $d \in D_\lightning$, we implicitly coerce it as the constant relation $D \bowtie \{d\}$. Hence, the top element of the $\mathbb{K}$ lattice is simply written $\lightning$. The notation $\uparrow f$ explicitly lifts function f from $D \to D$ to $\mathbb{K}$.

Given a relation $K \in \mathcal{R}(D, D_\lightning)$, we define its lifting $\lceil K$ in $\mathcal{R}(D_\lightning, D_\lightning)$ by $\lceil K \triangleq K \cup \{(\lightning, \lightning)\}$. This allows us to define the sequence of computations by $K_1 ; K_2 \triangleq K_1 \cdot \lceil K_2$, and the unbounded iteration of this sequence (i.e. a loop with a runtime-chosen number of iterations) by

$$K^* \triangleq (\lceil K)^* \cap (D \times D_\lightning)$$

Given a predicate $P \in \mathcal{P}(D)$, we define the notion of *assumption* (or *guard*) as $\dashv P \triangleq (P \bowtie D) \sqcap \varepsilon$. Informally, if P is satisfied on the current state then $\dashv P$ skips: it behaves like ε. Otherwise, $\dashv P$ produces no output: it behaves like $\bot$.

[9] A postcondition is thus in $\mathcal{P}(D \times D)$ instead of the original $\mathcal{P}(D)$: this standard generalization avoids introducing "auxiliary variables" to represent the input state.

[10] However, in our algebra, $\sqsubseteq$ corresponds to "*refines*", whereas in standard refinement calculus it dually corresponds to "*is refined by*". Actually, our convention follows lattice notations of abstract interpretation.

We also define the dual notion of *assertion* as $\vdash P \triangleq (\dashv \neg P\,;\lightning) \sqcup \varepsilon$. If P is not satisfied on the current state, then $\vdash P$ produces an error. Otherwise, it skips.

With these operators, $\mathbb{K}$ provides a convenient language to express specifications: any Hoare specification (P, Q) of $\mathcal{P}(D) \times \mathcal{R}(D, D)$ is expressed as the computation $\vdash P; Q$. Moreover, refinement allows to express usual Verification Conditions (VC) of Hoare Logic, for partial and total correctness. For our toy analyzer (described later), we only need VC for partial correctness. Typically, we use the usual partial correctness VC of unbounded iteration: K^* is equivalent to produce an output satisfying *every* inductive invariant I of K.

$$K^* \equiv \bigsqcap_{I \in \{I \in \mathcal{P}(D) \mid K \sqsubseteq \vdash I; D \rtimes I\}} \vdash I; D \rtimes I$$

In this equivalence, the $\sqsubseteq$-way corresponds to the soundness of the VC, whereas the $\sqsupseteq$-way corresponds to its completeness. In our context, such a soundness proof typically ensures that the specification of an abstract computation is refined by concrete semantics of the analyzed code. It guarantees that the analysis is correct w.r.t. semantics of the analyzed code.

Let us here make clear that if computation $\bot$ is naturally interpreted as "non-termination", it is also usual to see $\lightning$ as non-termination in total correctness. More formally, given any computation K, we express (weak)termination of K as the predicate $\mathrm{trm}(K) \triangleq \{d \mid \exists d' \in D, d \xrightarrow{K} d'\}$. Then, we can change non-termination of K into an error by using computation "$\vdash \mathrm{trm}(K)\,; K$" instead of K. This reduction of total correctness to partial correctness is standard in Hoare logic (e.g. verification conditions of loop variants).

Example on a Toy Language. Let t stands for an arithmetic term and c be a condition over numerical variables, whose syntax is $c ::= t_1 \bowtie t_2 \mid \neg c \mid c_1 \wedge c_2 \mid c_1 \vee c_2$ with $\bowtie \in \{=, \neq, \leq, \geq, <, >\}$. Semantics $[\![t]\!]$ of t and $[\![c]\!]$ of c work with a domain of integer memories $D \triangleq \mathbb{V} \to \mathbb{Z}$ where $\mathbb{V}$ is the type of variables. Hence, $[\![t]\!] \in D \to \mathbb{Z}$ and $[\![c]\!] \in \mathcal{P}(D)$. We omit their definition here.

Let us now introduce a small imperative programming language named $\mathbb{S}$ for which we will describe a toy analyzer in Section 7.2.2. The syntax of a $\mathbb{S}$ program s is described on Figure 7.1 together with its big-steps semantics $[\![s]\!]$ in $\mathbb{K}$. This semantics is defined recursively on the syntax of s using guarded commands derived from $\mathbb{K}$. First, we define $\dashv c \triangleq \dashv [\![c]\!]$ and $\vdash c \triangleq \vdash [\![c]\!]$. We also use command "$x := t$" defined as $\uparrow \lambda d.d[x := [\![t]\!](d)]$, where the memory assignment written "$d[x := n]$" – for $d \in D$, $x \in \mathbb{V}$ and $n \in \mathbb{Z}$ – is defined as the function $\lambda x' : \mathbb{V}, \texttt{if}\ x' = x\ \texttt{then}\ n\ \texttt{else}\ d(x')$.

s	**assert**(c)	$x \leftarrow t$	$s_1\,;s_2$	**if**$(c)\{s_1\}$**else**$\{s_2\}$	**while**$(c)\{s\}$
$[\![s]\!]$	$\vdash c$	$x := t$	$[\![s_1]\!]\,;[\![s_2]\!]$	$\dashv c\,;[\![s_1]\!] \sqcup \dashv \neg c\,;[\![s_2]\!]$	$(\dashv c\,;[\![s]\!])^*\,;\dashv \neg c$

Figure 7.1: Syntax and concrete semantics of $\mathbb{S}$

At this point, we have defined an algebra $\mathbb{K}$ of concrete computations: a language that we use to express specifications – for instance, in the form of Hoare specifications – on abstract computations. This algebra also provides denotations for defining big-steps semantics (like in Figure 7.1). Hence, $\mathbb{K}$ is aimed at providing an intermediate level between operational semantics of programs and their abstract interpretations (with the same purpose than the intermediate Hoare Logic in Verasco [Jou16]). The next section defines how we certify correctness of abstract computations w.r.t. $\mathbb{K}$ computations.

7.2.2 Composing Diagrams to Certify Abstract Computations

Rice's theorem states that the property $d \xrightarrow{K} d'$ is undecidable. In the theory of abstract interpretation [CC77], we approximate K by a *computable (terminating) function* $\sharp K$ working on an approximation $\sharp D$ of $\mathcal{P}(D)$. Set $\sharp D$ is called an *abstract domain* and it is related to $\mathcal{P}(D)$ by a concretization function $\gamma : \sharp D \to \mathcal{P}(D)$. Function $\sharp K$ is called an *abstract interpretation* (or *abstract computation*) of K. This chapter considers two abstract domains, intervals and convex polyhedra, associated with the concrete domain $D \triangleq \mathbb{V} \to \mathbb{Z}$ involved in Figure 7.1.

1. Given $\mathbb{Z}_\infty \triangleq \mathbb{Z} \uplus \{-\infty, +\infty\}$, an abstract memory $\sharp d$ of the interval domain is a finite map associating each variable x with an interval $[a_x, b_x]$ of $\mathbb{Z}_\infty \times \mathbb{Z}_\infty$. Its concretization is the set of concrete memory states satisfying the constraints of $\sharp d$, i.e.

$$\gamma(\sharp d) \triangleq \{d \in D \mid \forall x, a_x \leq d(x) \leq b_x\}$$

2. The concretization of a convex polyhedron $\sharp d = \bigwedge_i \sum_j a_{ij}.x_j \leq b_i$, where a_{ij}'s and b_i's are rational constants and x_j's are integer program variables, is

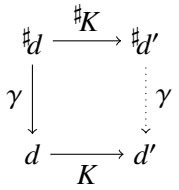
$$\gamma(\sharp d) \triangleq \{d \in D \mid \bigwedge_i \sum_j a_{ij}.d(x_j) \leq b_i\}$$

Correctness Diagrams of Impure Abstract Computations. Our framework only deals with partial correctness: we do not prove that abstract computations terminate, but only that they are a sound over-approximation of their corresponding concrete computation. Moreover, abstract computations may invoke untrusted oracles, whose results are verified by a certified checker. A bug in those oracles may make the whole computation nondeterministic or divergent. Thus, it is potentially unsound to consider abstract computations as pure functions. In this simplified presentation of our framework, we define abstract computations as relations in $\sharp\mathbb{K} \triangleq \mathcal{R}(\sharp D, \sharp D)$. A more elaborate representation – based on monads – is defined in Section 7.4.1, in order to extract abstract computations from Coq to OCaml functions. We express correctness of abstract computations through commutative diagrams defined as follows.

Definition 7.1 (Correctness of abstract computations)**.** An abstract computation $\sharp K$ of $\sharp\mathbb{K}$ is correct w.r.t. a concrete computation K of $\mathbb{K}$ iff

$\forall \sharp d, \sharp d' \in \sharp D, \quad \forall d \in D, \forall d' \in D_{\lightning}$,

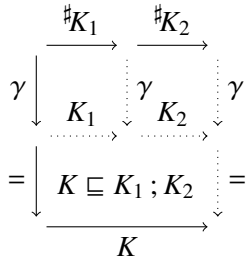
$$\sharp d \xrightarrow{\sharp K} \sharp d' \ \wedge \ d \xrightarrow{K} d' \ \wedge \ d \in \gamma(\sharp d) \qquad \Rightarrow \qquad d' \in \gamma(\sharp d')$$



Note that $d' \in \gamma(\sharp d')$ implies itself that $d' \neq \lightning$ because $\lightning$ is not in the image of γ.

Such a diagram thus corresponds to a pair of an abstract and a concrete computation, with a proof that the abstract one is correct w.r.t. the concrete one. As illustrated on the example below, these diagrams allow to build *compositional proofs* that an abstract computation, composed of several simpler parts, is correct w.r.t. a concrete computation. Diagrams are indeed preserved by several composition operators, and also by refinement of concrete computations.

As an example, consider two abstract computations $^\sharp K_1$ and $^\sharp K_2$ that are correct w.r.t. concrete K_1 and K_2. In order to show that the sequential composition $^\sharp K_1 \cdot {}^\sharp K_2$ is correct w.r.t. concrete K, it suffices to prove that $K \sqsubseteq K_1 \,;\, K_2$, as illustrated on the right hand side scheme.



In the following, we introduce a datatype written $^\dagger\mathbb{K}$ to represent these diagrams: a diagram $^\dagger K \in {}^\dagger\mathbb{K}$ represents an abstract computation $^\sharp K$ which is correct w.r.t. its associated concrete computation K. The core of our approach is to lift guarded-commands on $\mathbb{K}$ involved in Figure 7.1 to guarded-commands on $^\dagger\mathbb{K}$. For instance, our toy analyzer $^\sharp[\![s]\!]$ for s in $\mathbb{S}$ is defined similarly to $[\![s]\!]$ of Figure 7.1, but from $^\dagger\mathbb{K}$ operators instead of $\mathbb{K}$ ones. For a given diagram $^\dagger K$, we can prove the correctness of an abstract computation $^\sharp K$ w.r.t. a concrete computation K' simply by proving that $K' \sqsubseteq K$. In practice, such refinement proofs are simplified using a weakest-liberal-precondition calculus (see Section 7.4.2).

Our Interface of Abstract Domains. The (simplified) theory of our abstract domains is defined in Figure 7.2. This theory is not included in Verasco's one because it allows impure operators (operators are relations, and not pure functions). Besides its concretization function γ, an abstract domain $^\sharp D$ provides constants $^\sharp\top$ and $^\sharp\bot$, representing respectively predicate true and false. It also provides abstract computations $^\sharp\dashv c$ and $x {}^\sharp{:=} t$ of $\mathcal{R}(^\sharp D, {}^\sharp D)$, which are respectively correct w.r.t. concrete computations $\dashv c$ and $x := t$. It provides operator $^\sharp\sqcup$ of $\mathcal{R}(^\sharp D \times {}^\sharp D, {}^\sharp D)$, which over-approximates the binary union on $\mathcal{P}(D)$. At last, it provides inclusion test $^\sharp\sqsubseteq$ of $\mathcal{R}(^\sharp D \times {}^\sharp D, \texttt{bool})$.

$$D \subseteq \gamma(^\sharp\top) \qquad \gamma(^\sharp\bot) \subseteq \emptyset \qquad {}^\sharp d \xrightarrow{^\sharp\dashv c} {}^\sharp d' \;\Rightarrow\; \gamma(^\sharp d) \cap [\![c]\!] \subseteq \gamma(^\sharp d')$$

$$^\sharp d \xrightarrow{x {}^\sharp{:=} t} {}^\sharp d' \wedge d \in \gamma(^\sharp d) \Rightarrow d[x := [\![t]\!](d)] \in \gamma(^\sharp d')$$

$$(^\sharp d_1, {}^\sharp d_2) \xrightarrow{^\sharp\sqcup} {}^\sharp d' \Rightarrow \gamma(^\sharp d_1) \cup \gamma(^\sharp d_2) \subseteq \gamma(^\sharp d') \qquad (^\sharp d_1, {}^\sharp d_2) \xrightarrow{^\sharp\sqsubseteq} \texttt{true} \Rightarrow \gamma(^\sharp d_1) \subseteq \gamma(^\sharp d_2)$$

Figure 7.2: Correctness specifications of our abstract domains

Abstract Computations of Guarded-commands. We derive our guarded-commands on $^\dagger\mathbb{K}$ in a generic way from any abstract domain satisfying the interface of Figure 7.2. As explained above, we lift each $\mathbb{K}$ guarded-command appearing in Figure 7.1 into a $^\dagger\mathbb{K}$ guarded-command. This lifting is detailed in Figure 7.3: a $^\dagger\mathbb{K}$ operator has the same notation as its corresponding $\mathbb{K}$ operator and maps it to an abstract computation of $^\sharp\mathbb{K}$. The diagrammatic proof relating them is straightforward from correctness specifications given in Figure 7.2. We now detail the ideas behind this mapping.

Concrete commands $\dashv c$ and $x := t$ are trivially associated with $^\sharp\dashv c$ and $x {}^\sharp{:=} t$. Concrete command $K_1 \,;\, K_2$ is associated with $^\sharp K_1 \cdot \underline{^\sharp K_2}$ – where $\underline{^\sharp K_2}$ returns $^\sharp\bot$ if the current abstract state is included in $^\sharp\bot$, or runs $^\sharp K_2$ otherwise. Concrete $K_1 \sqcup K_2$ is lifted by applying operator $^\sharp\sqcup$ to the results of $^\sharp K_1$ and $^\sharp K_2$.

Concrete assertion $\vdash c$ is associated with checking that the result of $^\sharp\dashv \neg c$ is *included* in $^\sharp\bot$: otherwise, the abstract computation *fails*. In our refinement proofs of abstract computations, "*to fail*" means "*to give no result*". Hence, concrete $\lightning$ is associated with abstract computation $\emptyset$ (and concrete

$\bot$ is associated with $^\sharp\bot$). However, for our implementation of abstract computations in Section 7.4.1, "*to fail*" means "*to raise an alarm for the user*". In other words, our notion of correctness on abstract computations only gives some guarantee when no alarm is raised. In our formal proofs, we do not make distinction between an abstract computation that raises an alarm and an one that diverges.

At last, concrete K^* is associated with an abstract computation that invokes an untrusted oracle proposing an inductive invariant $^\sharp d_i$ of $^\sharp K$ for the current abstract state. Thus, using inclusion tests, $^\sharp(K^*)$ checks that $^\sharp d_i$ is actually an inductive invariant (otherwise, it fails), before returning it as the output abstract state.

$^\dagger\mathbb{K}$	Spec. in $\mathbb{K}$	Impl. in $^\sharp\mathbb{K}$
$\dashv c$	$\dashv c$	$^\sharp\dashv c$
$x := t$	$x := t$	$x^\sharp{:=}t$
$^\dagger K_1 \,;\, ^\dagger K_2$	$K_1 \,;\, K_2$	$^\sharp K_1.\{(^\sharp d_1, ^\sharp d_2) \mid \exists b, (^\sharp d_1, ^\sharp\bot) \overset{^\sharp\sqsubseteq}{\rightarrow} b \wedge$ **if** b **then** $^\sharp d_2 = {}^\sharp\bot$ **else** $^\sharp d_1 \overset{^\sharp K_2}{\longrightarrow} {}^\sharp d_2\}$
$^\dagger K_1 \sqcup {}^\dagger K_2$	$K_1 \sqcup K_2$	$\{(^\sharp d, ^\sharp d') \mid \exists ^\sharp d_1, \exists ^\sharp d_2, ^\sharp d \overset{^\sharp K_1}{\longrightarrow} {}^\sharp d_1 \wedge {}^\sharp d \overset{^\sharp K_2}{\longrightarrow} {}^\sharp d_2 \wedge (^\sharp d_1, ^\sharp d_2) \overset{^\sharp\sqcup}{\rightarrow} {}^\sharp d'\}$
$\vdash c$	$\vdash c$	$\{(^\sharp d, ^\sharp d) \mid \exists ^\sharp d', ^\sharp d \overset{^\sharp\dashv\neg c}{\longrightarrow} {}^\sharp d' \wedge (^\sharp d', ^\sharp\bot) \overset{^\sharp\sqsubseteq}{\rightarrow} \texttt{true}\}$
$^\dagger K^*$	K^*	$\{(^\sharp d, ^\sharp d_i) \mid (^\sharp d, ^\sharp d_i) \overset{^\sharp\sqsubseteq}{\rightarrow} \texttt{true} \wedge \exists ^\sharp d', ^\sharp d \overset{^\sharp K}{\rightarrow} {}^\sharp d' \wedge (^\sharp d', ^\sharp d_i) \overset{^\sharp\sqsubseteq}{\rightarrow} \texttt{true}\}$

Figure 7.3: Guarded-commands of $^\dagger\mathbb{K}$ involved in $\mathbb{S}$ analysis

7.2.3 Higher-order Programming with Correctness Diagrams

Our linearization procedure detailed in Section 7.3.2 illustrates how we use GCL $^\dagger\mathbb{K}$ as a programming language for abstract computations. GCL $\mathbb{K}$ is our specification language. Each program $^\dagger K$ of $^\dagger\mathbb{K}$ is associated with a specification K of $\mathbb{K}$ syntactically derived from its code through mapping of Figure 7.3 and Figure 7.4. Indeed, Figure 7.4 details two other operators of $^\dagger\mathbb{K}$ invoked by our linearization procedure. First, operator ($\texttt{cast}\ ^\dagger K\ K'$) casts a diagram $^\dagger K$ to a given specification K': it requires $K' \sqsubseteq K$ in order to produce a new valid $^\dagger\mathbb{K}$ diagram. This cast operator thus leads to a modular design of the certified development since it allows stepwise refinement of $^\dagger\mathbb{K}$ diagrams. Second, operator ($\pi\, ^\dagger{\gg\!=}_Q\, ^\dagger g$) – where, for a given type A, π is of type $\mathcal{R}(^\sharp D, A)$ and $^\dagger g$ of type $A \rightarrow {}^\dagger\mathbb{K}$ – binds the results of π to $^\dagger g$. This operator requires a *concrete* postcondition Q of $A \rightarrow \mathcal{P}(D)$ on the results of π (see Figure 7.4).

$^\dagger\mathbb{K}$	Spec. in $\mathbb{K}$	Impl. in $^\sharp\mathbb{K}$	Under precondition
$\texttt{cast}\ ^\dagger K\ K'$	K'	$^\sharp K$	$K' \sqsubseteq K$
$\pi\, ^\dagger{\gg\!=}_Q\, ^\dagger g$	$\bigsqcap_x \vdash Q\,x\,;\, g\,x$	$\{(^\sharp d_1, ^\sharp d_2) \mid \exists x, ^\sharp d_1 \overset{\pi}{\rightarrow} x \wedge {}^\sharp d_1 \overset{^\sharp g\,x}{\longrightarrow} {}^\sharp d_2\}$	$\forall ^\sharp d, \forall x \in A,$ $^\sharp d \overset{\pi}{\rightarrow} x \Rightarrow \gamma(^\sharp d) \subseteq Q\,x$

Figure 7.4: $^\dagger\mathbb{K}$ operators that generate proof obligations

More specifically, Section 7.3.2 applies our refinement calculus to certify higher-order abstract computations. Indeed, our linearization procedure *partitions* abstract states in order to increase preci-

sion. Continuation-Passing-Style (CPS) [Rey93] is a higher-order pattern that provides a lightweight and modular style to program and certify simple partitioning strategies. Let us now detail this idea.

Given an abstract state $^\sharp d$, our linearization procedure invokes a sub-procedure $^\sharp f$ that splits $^\sharp d$ into a partition $(^\sharp d_i)_{i\in I}$ and computes a value r_i (of a given type A) for each cell $^\sharp d_i$. Then, the linearization procedure *continues* the computation from each cell $(r_i, ^\sharp d_i)$ to finally return the join of all cells. In other words, from $^\sharp d$, $^\sharp f$ computes $(r_i, ^\sharp d_i)_{i\in I}$. The main procedure finally computes $^\sharp\bigsqcup_{i\in I}(^\sharp g\ r_i\ ^\sharp d_i)$ – where $^\sharp g$ is a given function of $A \to {}^\sharp\mathbb{K}$. In order to avoid explicit handling of partitions, we make $^\sharp g$ a parameter of $^\sharp f$ to perform the join inside $^\sharp f$. In this style, $^\sharp f$ is of type $(A \to {}^\sharp\mathbb{K}) \to {}^\sharp\mathbb{K}$ and the parameter $^\sharp g$ of $^\sharp f$ is called their *continuation*.

However, specifying directly the correctness of computations that use CPS is not obvious because of the higher-order parameter. Actually, we define $^\dagger f$ of type $(A \to {}^\dagger\mathbb{K}) \to {}^\dagger\mathbb{K}$ and work with a continuation $^\dagger g$ of type $A \to {}^\dagger\mathbb{K}$. This allows us to specify CPS abstract computations w.r.t. CPS concrete computations. An example of such a specification is detailed later in Figure 7.8. Therefore, we keep implicit the notion of partition, both in specification and in implementation.

Similarly, CPS enables to implement some dynamic strategies of trace partitioning[MR05]. In abstract interpretation, "*trace partitioning*" corresponds to partition the set of all possible *execution traces* of the analyzed program in order to improve accuracy. Controlling the partitioning process is motivated by the fact that $(^\sharp K_1 \cdot {}^\sharp K_3)^\sharp\sqcup(^\sharp K_2 \cdot {}^\sharp K_3) \,^\sharp\sqsubseteq\, (^\sharp K_1{}^\sharp\sqcup{}^\sharp K_2) \cdot {}^\sharp K_3$, but the opposite inclusion does not hold. Hence, the left side is more precise whereas the right one is faster, as computation $^\sharp K_3$ is factorized. In practice, *dynamic* trace partitioning strategies select one of these two alternatives according to information of the current abstract state. The trace partitioning domain of [MR05] provides a functor able to extend a given abstract domain with dynamic partitioning management. More modestly, CPS allows for selecting some trace partitioning strategy at each function call, through the choice of its continuation. For instance, we define $^\dagger f \triangleq \lambda^\dagger g, (^\dagger g\ ^\dagger K_1) \sqcup (^\dagger g\ ^\dagger K_2)$. Then, the precise alternative derives from $^\dagger f\ \lambda^\dagger K, (^\dagger K\,;{}^\dagger K_3)$ whereas the fast one derives from $(^\dagger f\ \lambda^\dagger K, {}^\dagger K)\,;{}^\dagger K_3$. The CPS approach has the advantage to be very lightweight: there is no need to define and certify a data-structure to manage partitions. But it is less expressive than a trace partitioning domain. Indeed, a trace partitioning domain provides two kinds of partitioning operations: one to split partitions and one to merge partitions. Thus, the decision of merging partitions is quite independent of the decision to split partitions. On the contrary, with CPS, there is a single decision for each split/merge pair. Hence, a trace partitioning domain enables more dynamic merging strategies.

7.3 Interval-based Linearization Strategies for Polyhedra

Initially in [#FB14], the {VPL} worked with affine terms given by the abstract syntax below where x is a variable and n a constant of $\mathbb{Z}$

$$t ::= n \mid x \mid t_1 + t_2 \mid n.t$$

We now explain how we have extended {VPL} operators of Figure 7.2 to support polynomial terms, where the product "$n.t$" is generalized into "$t_1 \times t_2$".

The {VPL} derives assignment operator $^\sharp{:=}$ from guard $^\sharp\!\!\rightarrow$ and two low-level operators: projection (as defined in Section 4.2) and renaming (see [#FB14]). It also derives the guard operator from a restricted one where conditions have the form $0 \bowtie t$ with $\bowtie \in \{\leq, =, \neq\}$. Hence, we only need to linearize the restricted guard $^\sharp\!\!\rightarrow 0 \bowtie p$, where p is a polynomial. Below, we use letter p for polynomials and only keep letter t for affine terms.

Roughly speaking, we approximate a guard $^\sharp\!\!\rightarrow 0 \bowtie p$ by guards $^\sharp\!\!\rightarrow 0 \bowtie [t_1, t_2]$ – where t_1 and t_2 are affine or infinite bounds – such that, in the current abstract state, $p \in [t_1, t_2]$. Approximated guards

$\natural 0 \bowtie [t_1, t_2]$ are defined by cases on $\bowtie$:

$\bowtie$	$\leq$	$=$	$\neq$
$\natural\ 0 \bowtie [t_1, t_2]$	$\natural\ 0 \leq t_2$	$\natural\ 0 \leq t_2 \wedge t_1 \leq 0$	$\natural\ 0 < t_2 \vee t_1 < 0$

Remark that $\natural\ t_1 \bowtie t_2$ simply performs the polyhedral reduction described in Section 6.1: it turns a polyhedron $(P \wedge t_1 \bowtie t_2)$ into a *reduced* polyhedron P' such that $(P \wedge t_1 \bowtie t_2) \Rightarrow P'$. This also applies to conjunctions of linear inequalities like in $\natural\ 0 \leq t_2 \wedge t_1 \leq 0$. Operation $\natural\ 0 \leq t_2 \wedge t_1 \leq 0$ itself corresponds to a convex-hull (Section 4.3) after two such polyhedral reductions.

We compute affine intervals "$[t_1, t_2]$" using heuristics inspired from [Min06], except that in order to increase precision, we dynamically partition the abstract state according to the sign of some affine subterms. This process will be detailed further.

Our certified linearization is built on a FVDP architecture: an untrusted oracle uses heuristics to select linearization strategies and a certified procedure applies them to build a correct-by-construction result. These strategies, which are listed in Section 7.3.1, allow to finely tune the precision-versus-efficiency trade-off of the linearization. Section 7.3.2 details the design of our oracle and illustrates our lightweight handling of partitions using CPS in our certified procedure.

7.3.1 Our List of Interval-Based Strategies

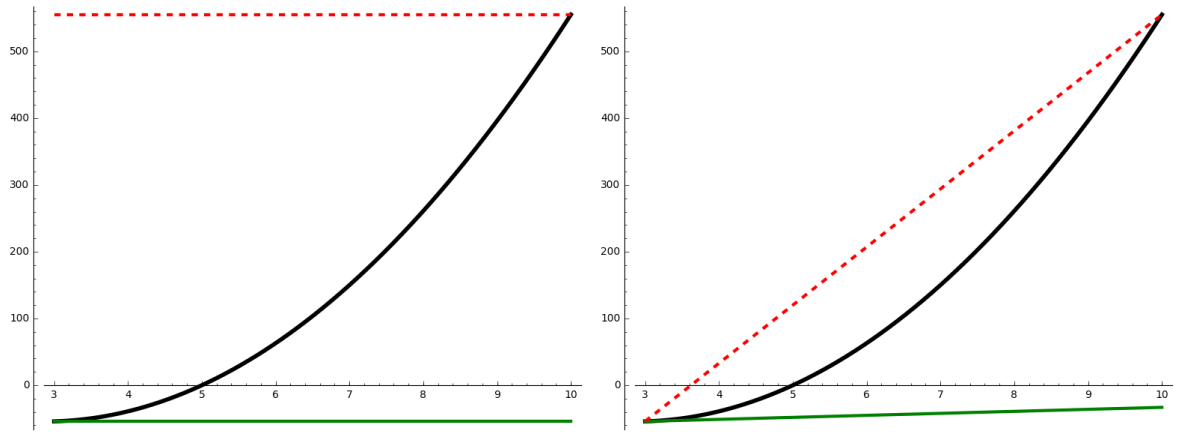
Constant Intervalization. Our fastest strategy applies a constant intervalization operator of the abstract domain. Given a polynomial p, this operator, written $\natural\pi(p)$, over-approximates p by an interval where affine terms are reduced to constants. More formally, $\natural\pi(p)$ is a computation of $\mathcal{R}(\natural D, \mathbb{Z}^2_\infty)$ such that if $\natural d \xrightarrow{\natural\pi(p)} [n_1, n_2]$, then $\gamma(\natural d) \subseteq \{d \mid n_1 \leq [\![p]\!]\, d \leq n_2\}$. It uses a naive interval domain, where arithmetic operations $+$ and $\times$ are approximated by their correspondence on intervals:
$[n_1, n_2] + [n_3, n_4] \triangleq [n_1 + n_3, n_2 + n_4]$, and
$[n_1, n_2] \times [n_3, n_4] \triangleq [\min(E), \max(E)]$ where $E = \{n_1.n_3, n_1.n_4, n_2.n_3, n_2.n_4\}$.

Example 7.3 (Constant intervalization)**.** On $x \in [3, 10]$, constant intervalization of $(3.x-15)\times(4.x-3)$ gives $(3.[3, 10]-15)\times(4.[3, 10]-3) = ([9, 30]-15])\times([12, 40]-3) = [-6, 15]\times[9, 37] = [-54, 555]$, as shown on Figure 7.5(a).

Ring Rewriting. A weakness of operator $\natural\pi$ is its sensitivity to ring rewriting. For instance, consider a polynomial p_1 such that $\natural\pi(p_1)$ returns $[0, n]$, $n \in \mathbb{N}^+$. Then $\natural\pi(p_1 - p_1)$ returns $[-n, n]$ instead of the precise result 0. Such imprecision occurs in barycentric computations such as $p_2 \triangleq p_1 \times t_1 + (n - p_1) \times t_2$ where affine terms t_1, t_2 are bounded by $[n_1, n_2]$. Indeed $\natural\pi(p_2)$ returns $2n.[n_1, n_2]$ instead of $n.[n_1, n_2]$. Moreover, if we rewrite p_2 into an equivalent polynomial $p'_2 \triangleq p_1 \times (t_1 - t_2) + n.t_2$, then $\natural\pi(p'_2)$ returns $n.[2.n_1 - n_2, 2.n_2 - n_1]$. If $n_1 > 0$ or $n_2 < 0$, then $\natural\pi(p'_2)$ is strictly more precise than $\natural\pi(p_2)$. The situation is reversed otherwise. Consequently, our oracle begins by simplifying the polynomial before trying to factorize it conveniently. But as illustrated above, it is difficult to find a factorization minimizing $\natural\pi$ results. We give more details on the ring rewriting heuristics of our oracle in the following.

Sign Partitioning. In order to find more precise bounds of polynomial p than those given by $\natural\pi(p)$, we look for an interval of two affine terms $[t_1, t_2]$ bounding p. Assume p is of the form $p' \times t$, with t an affine term and p' a polynomial. Let $[n'_1, n'_2]$ be the constant intervalization of p' obtained from $\natural\pi(p')$. Depending on the sign of t, we deduce affine bounds of p in the following way:

- if $0 \leq t$, then $p' \times t \in [n'_1.t, n'_2.t]$



(a) Constant intervalization **(b)** Focusing & affine intervalization

Figure 7.5: Two intervalizations of $p = (3.x - 15) \times (4.x - 3)$ with $x \in [3, 10]$. Constant intervalization leads to $p \in [-54, 555]$, whereas focusing gives $p \in [3.x - 63, 87.x - 315]$.

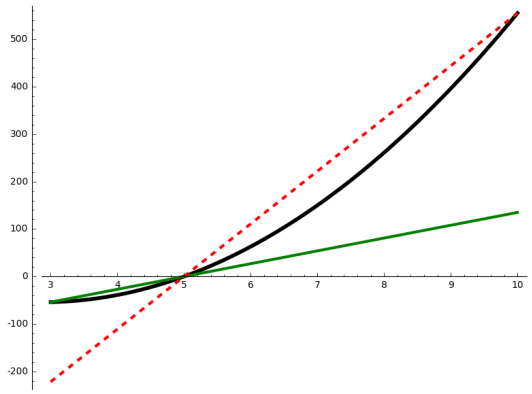


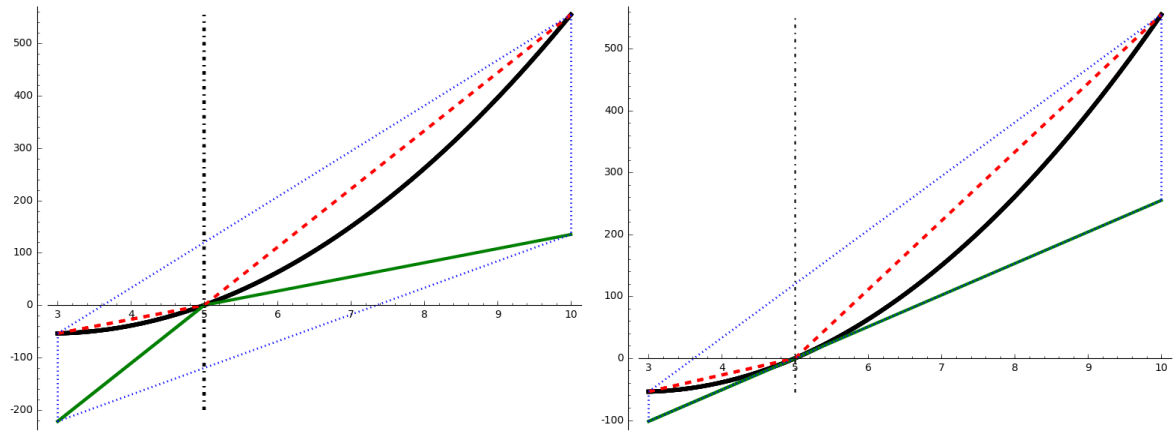
Figure 7.6: A wrong affine intervalization of $p = (3.x - 15) \times (4.x - 3)$ with $x \in [3, 10]$.

- if $t < 0$, then $p' \times t \in [n'_2.t, n'_1.t]$

When the sign of t is known, we discard one of these two cases and thus have a fast affine approximation of $p' \times t$. This is the case in Figure 7.5(b) (the underlying computations are detailed in Example 7.6). When the sign of t is unknown, we split the analysis for each sign of t.

More generally, we split the current abstract state $\sharp d$ into a partition $(\sharp d_i)_{i \in I}$ according to the sign of some affine subterms of polynomial p, such that each cell $\sharp d_i$ leads to its own affine interval $[t_{i,1}, t_{i,2}]$. Finally, $\sharp \lrcorner 0 \bowtie p$ is over-approximated by computing the join of all $\sharp \lrcorner 0 \bowtie [t_{i,1}, t_{i,2}]$. The main drawback of sign partitioning is a worst-case exponential blow-up if applied systematically.

Example 7.4 (Sign partitioning)**.** Consider $p = (4.x-3)\times(3.x-15)$ with $x \in [3, 10]$, as in Example 7.3. First, we compute the constant intervalization of the left term $(4.x - 3)$, which gives $p = (4.[3, 10] - 3) \times (3.x - 15) = ([12, 40] - 3) \times (3.x - 15) = [9, 37] \times (3.x - 15) = [9.(3.x - 15), 37.(3.x - 15)]$. We obtain the two affine terms $9.(3.x - 15)$ and $37.(3.x - 15)$. But as shown on Figure 7.6, for $x \in [3, 10]$, these two terms are not comparable. Indeed, $9.(3.x - 15)$ is not always lower than $37.(3.x - 15)$ on $x \in [3, 10]$. Thus, $[9.(3.x - 15), 37.(3.x - 15)]$ is not a well-defined interval of affine forms. In order



(a) Static partitioning **(b)** Dynamic partitioning

Figure 7.7: Sign partitioning of $p = (3.x - 15) \times (4.x - 3)$ with $x \in [3, 10]$. Static partitioning gives $p \in [51.x - 375, 87.x - 315]$, whereas dynamic one gives $p \in [51.x - 255, 87.x - 315]$.

to get an affine interval bounding p, we need to partition the space at the point where these two terms are equal, i.e. at the point where $3.x - 15 = 0$ which is $x = 5$. Then, by intervalizing in both cells $3.x - 15 < 0$ and $3.x - 15 \geq 0$, we get:

$$p = \begin{cases} \left[\!\left[\; 37.(3.x - 15) \; , \;\; 9.(3.x - 15) \; \right]\!\right] & \text{if } 3.x - 15 < 0 \\ \left[\!\left[\; 9.(3.x - 15) \; , \;\; 37.(3.x - 15) \; \right]\!\right] & \text{if } 3.x - 15 \geq 0 \end{cases}$$

The result is shown on Figure 7.7(a). To obtain the final result of the linearization, it is necessary to compute the convex hull of both sides. Here, the result is $p \in [51.x - 375, 87.x - 315]$, and it appears as the blue dotted polyhedron on the figure.

Let us also illustrate sign partitioning for the previous barycentric-like computation of p_2'. By convention, our certified procedure partitions the sign of right affine subterms (here, the sign of $t_1 - t_2$). Hence, it founds $p_2' \in [n.t_2, n.t_1]$ in cell $0 \leq t_1 - t_2$, and $p_2' \in [n.t_1, n.t_2]$ in cell $t_1 - t_2 < 0$. When it joins the two cells, $\sharp_4 0 \bowtie p_2'$ is computed as $\sharp_4 0 \bowtie n.[n_1, n_2]$ as we expect for such a barycentre. Note that sign partitioning is also sensitive to ring rewriting. In particular, the oracle may rewrite a product of affine terms $t_1 \times t_2$ into $t_2 \times t_1$, in order to discard t_1 instead of t_2 by sign partitioning.

Static vs Dynamic Intervalization During Partitioning. Computing the constant bounds of an affine term inside a given polyhedron invokes a costly linear programming procedure. Hence, for a given polynomial p to approximate, we start by computing an environment σ that associates each variable of p with a constant interval: as detailed later, this environment is indeed used by heuristics of our oracle. By default, operator $\sharp_\pi$ is called in *dynamic* mode, meaning that each bound is computed dynamically in the current cell – generated from sign partitioning – using linear programming. If one wants a faster use of operator $\sharp_\pi$, he may invoke it in *static* mode, where bounds are computed using σ.

For instance, let us consider the sign partitioning of $p \triangleq t_1 \times t_2$ in the context $0 < n_1, n_2$ and $-n_1 \leq t_2 \leq t_1 \leq n_2$. In cell $0 \leq t_2$, static mode bounds p by $[-n_1.t_2, n_2.t_2]$, whereas dynamic mode bounds p by $[0, n_2.t_2]$. In cell $t_2 < 0$, both modes bound p by $[n_2.t_2, -n_1.t_2]$. On the join of these cells, both modes give the same upper bound. But the lower bound is $-n_1.n_2$ for static mode, whereas it is $\frac{n_1.n_2}{n_1+n_2}(t_2 + n_1) - n_1.n_2$ for dynamic mode, which is strictly more precise.

Example 7.5 (Static vs Dynamic Intervalization)**.** In Example 7.4, we saw that partitioning on the sign of $(3.x - 15)$ gave

$$\begin{aligned} p &= (4.x - 3) \times (3.x - 15) \\ &= [9, 37] \times (3.x - 15) \\ &= \begin{cases} \left[\, 37.(3.x - 15)\,,\ \ 9.(3.x - 15) \,\right] & \text{if } 3.x - 15 < 0 \\ \left[\, 9.(3.x - 15)\ \,,\ 37.(3.x - 15) \,\right] & \text{if } 3.x - 15 \geq 0 \end{cases} \end{aligned}$$

This intervalization is in fact a *static* one because $(4.x - 3)$ was intervalized in the same way in both cells, using $x \in [3, 10]$. Instead, using *dynamic* intervalization during partitioning will improve the precision by finding better bounds of $(4.x - 3)$. Indeed, building on the fact that $x \in [3, 5]$ on cell $(3 \leq x \wedge 3.x - 15 < 0)$, intervalizing $(4.x - 3)$ gives $(4.[3, 5] - 3) = ([12, 20] - 3) = [9, 17]$. Similarly, on cell $(x \leq 10 \wedge 3.x - 15 \geq 0)$, $x \in [5, 10]$ hence an intervalization of $(4.x - 3)$ by $(4.[5, 10] - 3) = ([20, 40] - 3) = [17, 37]$. Thus,

$$\begin{aligned} p &= \begin{cases} [9, 17] \times (3.x - 15) & \text{if } 3.x - 15 < 0 \\ [17, 37] \times (3.x - 15) & \text{if } 3.x - 15 \geq 0 \end{cases} \\ &= \begin{cases} \left[\, 17.(3.x - 15)\,,\ \ 9.(3.x - 15) \,\right] & \text{if } 3.x - 15 < 0 \\ \left[\, 17.(3.x - 15)\,,\ 37.(3.x - 15) \,\right] & \text{if } 3.x - 15 \geq 0 \end{cases} \end{aligned}$$

As explained before, the final result is obtained by computing the convex hull of both cells. Here, we get $p \in [51.x - 255, 87.x - 315]$. The difference between static and dynamic partitioning is shown on Figure 7.7. We can see that the lower bound of p has been significantly improved by the dynamic partition. The upper bound resulting from static partitioning was already optimal.

Focusing. Focusing is a ring rewriting heuristic that may increase the precision of sign partitioning. Given a product $p \triangleq t_1 \times t_2$, we define the *focusing* of t_2 in center n as the rewriting of p into $p' \triangleq n.t_1 + t_1 \times (t_2 - n)$. Thanks to this focusing, the affine term $n.t_1$ appears whereas t_1 would otherwise be discarded by sign partitioning. Let us simply illustrate the effect of this rewriting when $0 \leq n \leq n'_1$ with t_1 (resp. t_2) bounded by $[n_1, n_2]$ (resp. $[n'_1, n'_2]$). Sign partitioning bounds p in affine interval $[n_1.t_2, n_2.t_2]$ whereas p' is bounded by interval $[n_1.t_2 + n.(t_1 - n_1), n_2.t_2 - n.(n_2 - t_1)]$. The former contains the latter since $t_1 - n_1$ and $n_2 - t_1$ are nonnegative. Under these assumptions, the precision is maximal when $n = n'_1$.

Applied carelessly, focusing may also decrease the precision. Consequently, on products $p'' \times t_2$, our oracle uses the following heuristic, which cannot decrease the precision: if $0 \leq n'_1$, then focus t_2 in center n'_1; if $n'_2 \leq 0$, then focus t_2 in center n'_2; otherwise, do not change the focus of t_2.

Example 7.6 (Focusing)**.** Consider $p = (3.x - 15) \times (4.x - 3)$ with $x \in [3, 10]$, as in previous examples. The focusing of term $(4.x - 3)$ on $4.3 - 3 = 9$ is $p' = 9.(3.x - 15) + (3x - 15) \times (4.x - 12)$. Affine intervalization of p' is done by sign partitioning of $(4.x - 12)$, where cell $4.x - 12 < 0$ is empty. Finally, by intervalization,

$$\begin{aligned} p &= 9.(3.x - 15) + (3x - 15) \times (4.x - 12) \\ &= (27.x - 135) + (3.[3, 10] - 15) \times (4.x - 12) \\ &= (27.x - 135) + [-6, 15] \times (4.x - 12) \\ &= (27.x - 135) + [-24.x + 72, 60.x - 180] \\ &= [3.x - 63, 87.x - 315] \end{aligned}$$

Figure 7.5(b) shows its result. Intervalizations of Figure 7.5(a) and of Figure 7.5(b) have similar running times, but this latter gives strictly more precise results. Intervalizations of Figures 7.7(b) and 7.5(b) are not comparable: Figure 7.7(b) is more precise on a significant part of the domain $x \in [3, 10]$, but Figure 7.5(b) is better around the lower-left corner. The precision of Figure 7.7(b) comes at a cost: it requires two constant intervalizations and a convex-hull instead of one single constant intervalization.

Conjunction of strategies. As we saw by comparing Figures 7.5(b) and 7.7(b), two distinct linearization strategies may lead to incomparable polyhedra. Here, we can improve precision by computing the intersection of these polyhedra. In our stepwise refinement approach, this corresponds indeed to remark that $\dashv c \sqsubseteq (\dashv c\,; \dashv c)$, and to implement each of these guards $\dashv c$ with a distinct linearization strategy. Let us remark here that a sequence of two strategies gives more precise results than intersecting independent runs of these strategies: the second one may benefit from informations discovered by the first one. This is illustrated in Example 7.7 below. We use this trick in order to ensure that our linearization necessarily improves and benefits from results of a naive but quick constant intervalization.

7.3.2 Design of our Implementation

We now describe our procedure in detail. For a guard $\sharp\!\dashv 0 \bowtie p$, our certified procedure first rewrites p into $p' + t$ where t is an affine term and p' a polynomial. This may keep the non-affine part p' small compared to the affine one t. Typically, if p' is syntactically equal to zero, we simply apply the standard affine guard $\sharp\!\dashv 0 \bowtie t$. Otherwise, we compute environment σ for p' variables. Then, we compute $\sharp\!\dashv 0 \bowtie [n_1 + t, n_2 + t]$ where $[n_1, n_2]$ is the result of $\sharp\pi(p')$ for static environment σ. As mentioned earlier, this ensures that the resulting linearization necessarily improves and benefits from this first constant intervalization. In particular, if this guard is unsatisfiable at this point, the rest of the procedure is skipped. Otherwise, we invoke our external oracle on p' and σ. This oracle returns a polynomial p'' enriched with tags on subexpressions. We handle three tags to direct the intervalization: `AFFINE` expresses that the subexpression is affine; `STATIC` expresses that the subexpression has to be intervalized in static mode; `INTERV` expresses that intervalization is done using only $\sharp\pi$ (instead of sign partitioning). At last, a special tag `SKIP_ORACLE` inserted at the root of p'' indicates that it is not worth attempting to improve naive constant intervalization (e.g. because p' is a too big polynomial, any attempt would be too costly). After that, when this special tag is absent, our certified procedure checks that $p' = p''$ using a normalization procedure defined in the standard distribution of Coq [GM05]. If $p' \neq p''$, our procedure simply raises an error corresponding to a bug in the oracle. If $p' = p''$, it invokes a CPS affine intervalization of p'' for continuation $\lambda[t_1, t_2], \dashv 0 \bowtie [t_1 + t, t_2 + t]$. The next paragraphs detail this certified CPS intervalization and then, our external oracle.

Certified CPS Affine Intervalization. We implement and prove our affine intervalization using the CPS technique described in Section 7.2.3. On polynomial p'' and continuation ${}^{\dagger}g$, the specification of our CPS intervalization is

$$\varepsilon \sqcap \bigsqcap_{[t_1,t_2]} \vdash \{d \mid t_1 \leq [\![p'']\!]\, d \leq t_2]\}\,; g[t_1, t_2]$$

The ε case corresponds to a failure of our procedure: typically, a subexpression is not affine as claimed by the external oracle. In case of success, the procedure selects nondeterministically some affine intervals $[t_1, t_2]$ bounding p'' before merging continuations on them. The procedure is implemented

Given $({}^\dagger\pi\ p)$ of $(\mathbb{Z}_\infty \times \mathbb{Z}_\infty \to {}^\dagger\mathbb{K}) \to {}^\dagger\mathbb{K}$ defined by $({}^\dagger\pi\ p)({}^\dagger g_0) \triangleq {}^\sharp\pi(p)\ {}^\dagger{\gg\!=}_{\lambda[n_1,n_2],\{d \mid n_1 \le [\![p]\!]\, d \le n_2\}}\ {}^\dagger g_0$ the ${}^\dagger\mathbb{K}$ program on the right-hand side satisfies the specification below:

$$\Big\lceil\Big\rceil_{[t_1,t_2]} \vdash \{d \mid t_1 \le [\![p \times t]\!]\, d \le t_2\}\,;\, g[t_1,t_2]$$

if static **then**
  $({}^\dagger\pi\ p)(\lambda[n_1,n_2],\ (\dashv 0 \le t\,;\, {}^\dagger g[n_1.t, n_2.t])$
    $\sqcup\ (\dashv t < 0\,;\, {}^\dagger g[n_2.t, n_1.t]))$
else
    $(\dashv 0 \le t\,;\, ({}^\dagger\pi\ p)(\lambda[n_1,n_2], {}^\dagger g[n_1.t, n_2.t]))$
  $\sqcup$ $(\dashv t < 0\,;\, ({}^\dagger\pi\ p)(\lambda[n_1,n_2], {}^\dagger g[n_2.t, n_1.t]))$

Figure 7.8: Sign partitioning for $p \times t$ with continuation ${}^\dagger g$

recursively over the syntax of p''. Figure 7.8 sketches the implementation and the specification of the sign partitioning subprocedure. The figure deals with a particular case where p'' is a polynomial written $p \times t$ with t affine. In the implementation part, Boolean `static` indicates the mode of ${}^\sharp\pi$. In static mode, we indeed factorize the computation of ${}^\sharp\pi$ on both cells of the partition.

Our linearization procedure is written in around 2000 Coq lines, proofs included. Among them, the CPS procedure and its subprocedures take only 200 lines. The bigger part – around 1000 lines – is thus taken by arithmetic operators on interval domains (constant and affine intervals).

Design of Our External Oracle. Our external oracle ranks variables according to their priority to be discarded by sign partitioning. Then, it factorizes variables with the highest priority. The priority rank is mainly computed from the size of intervals in the precomputed environment σ: unbounded variables must not be discarded whereas variables bounded by a singleton are always discarded by static intervalization. Our oracle also tries to minimize the number of distinct variables that are discarded: variables appearing in many monomials have a higher priority. The oracle also interleaves factorization with focusing. Our oracle is written in 1300 lines of OCaml code.

Example 7.7 (A full run of the certified procedure)**.** Let us consider the effect of our linearization procedure on guard $\dashv x \times (y-2) \le z$ in a context where $(0 \le x) \wedge (x+1 \le y \le 1000) \wedge (z \le -2)$. First, note that a constant intervalization of $z - x \times (y-2)$ would bound it in $]-\infty, 997]$, and thus would not deduce anything useful from this guard.

Instead, our procedure rewrites the guard into $\dashv 0 \le p' + t$ with $p' \triangleq -x \times y$ and $t \triangleq z + 2x$. Then, it computes environment $\sigma \triangleq \{x \mapsto [0,999], y \mapsto [1,1000]\}$ and applies constant intervalization on p', leading to $p' \in]-\infty, 0]$. As you may notice, approximating this guard requires only an upper-bound on p', and our procedure does not compute the useless lower bound. From this first approximation of $\dashv 0 \le p' + t$, it deduces $0 \le t$.

Then, our oracle, invoked on p' and σ, decides to focus y in center 1 and thus rewrites p' as $p'' \triangleq x \times (1-y) - x$. Here, our CPS subprocedure only intervalizes the nonlinear part $x \times (1-y)$ using sign partitioning on $1-y$. Because we know $1 \le x$ from $2 \le -z \le 2x$, we deduce $1 - y \le -x \le -1$. Therefore, because $1 - y < 0$ and $1 \le x$, sign partitioning on $1-y$ bounds $x \times (1-y)$ by $]-\infty, 1-y]$. At last, CPS intervalization now approximates $\dashv 0 \le p' + t$ in $\dashv 0 \le 1 - y - x + t$. In fact, this implies $0 \le z$ which contradicts $z \le -2$. Hence, our polyhedral approximation of $\dashv x \times (y-2) \le z$ detects that this guard is unsatisfiable in the given context.

As a conclusion, let us remark that the first approximation leading to $0 \le t$ is necessary to the full success of the second one.

7.4 A Lightweight Refinement Calculus in Coq

Our implementation in Coq reformulates Section 7.2 with more computational representations of binary relations. Section 7.4.1 presents the representation change of impure abstract computations. Section 7.4.2 presents that of concrete ones. Finally, Section 7.4.3 presents our datatypes for correctness diagrams of abstract computations. Sections 7.4.1 and 7.4.3 also detail how the framework is adapted in order to handle alarms during the analysis.

7.4.1 Efficient Representations of Impure Abstract Computations

In the {VPL}, all computations involving oracles (including abstract computations) are "impure" – in the sense defined in Section 2.2 This section links this representation to relations from Section 7.2.

A relation R of $\mathcal{R}(A, B)$ can be equivalently seen as the function $\lambda x, \{y \mid x \overset{R}{\to} y\}$ of $A \to \mathcal{P}(B)$. This curryfied representation is the basis of {VPL} representation for impure computations, where type "$\mathcal{P}(B)$" is axiomatized in Coq as type "$??B$" as presented in Section 2.2.1. Indeed, this technique provides a Coq representation of relations that can be turned into an OCaml function at extraction.

Hence, all impure computations of $\mathcal{R}(A, B)$ in Figure 7.2 are actually expressed in our Coq development as functions of $A \to ??B$ in a given may-return monad. Indeed, the interface of may-return monads also allows to hide data-structure details – such as handling of alarms – for the correctness proof of abstract computations. The next paragraphs detail these ideas.

Correspondence with Set Theory Notations of Section 7.2. We recall that the abstraction of set "$\mathcal{P}(A)$" as type "$??A$" is given by the following definitions:

$$??A \triangleq \mathcal{P}(A) \qquad k \rightsquigarrow a \triangleq a \in k \qquad \varepsilon\, a \triangleq \{a\} \qquad k_1 \gg\!= k_2 \triangleq \bigcup_{a \in k_1} (k_2\, a)$$

Conversely, for any may-return monad, a computation k of $A \to ??B$ represents a relation of $\mathcal{R}(A, B)$ defined by $d \overset{k}{\to} d' \triangleq k\, d \rightsquigarrow d'$. Given two abstract computations k_1 and k_2 in ${}^\sharp D \to ??{}^\sharp D$, then "$\lambda x, (k_1\ x) \gg\!= k_2$" corresponds to a subrelation of "$k_1 \cdot k_2$".

Impure Computations of the Core May-return Monad. The {VPL} is parametrized by a *core* may-return monad that axiomatizes external computations (i.e. an ancester of the {Impure} library). This monad avoids a potential cause of unsoundness by expressing that external oracles are not pure functions, but encode relations. It is instantiated at extraction by providing the identity implementation given in Figure 2.2.

Of course, this implementation of the core monad remains hidden for our Coq proofs: they are thus valid for any instance of a may-return monad.

Alarm Handling in the Analyzer. Our toy analyzer, specified in Figure 7.1, handles alarms in the style of Verasco [Jou16]. On a potential error, it does not stop its analysis, but writes an alarm – represented here as a value of type `alarm` – and continues the analysis. Technically, this corresponds to lifting the core monad through a *writer monad transformer* [LH96]. Actually, we assume that the core monad has already an operation to write alarms ${}^c\texttt{write} : \texttt{alarm} \to {}^c??\texttt{unit}$, which is efficiently extracted as OCaml external code. On the Coq side, our *alarm writer monad* thus only encodes the underlying list of alarms as a Boolean: `true` corresponds to an empty list of alarms. It is defined in Figure 7.9 where alarm writer (resp. core) constructs are prefixed by a "^{w}" (resp. "^{c}"). The

$$^{w}??A \triangleq {}^{c}??(A \times \mathtt{bool}) \qquad k \,{}^{w}\!\rightsquigarrow a \triangleq k \,{}^{c}\!\rightsquigarrow (a, \mathtt{true}) \qquad {}^{w}\varepsilon\, a \triangleq {}^{c}\varepsilon\,(a, \mathtt{true})$$

$$k_1 \,{}^{w}\!\gg\!= k_2 \triangleq k_1 \,{}^{c}\!\gg\!= \lambda(a_1, l_1), (k_2\, a_1)\,{}^{c}\!\gg\!= \lambda(a_2, l_2), {}^{c}\varepsilon\,(a_2, l_1 \wedge l_2)$$

$$\mathtt{lift}\, k \triangleq k \,{}^{c}\!\gg\!= \lambda a, {}^{c}\varepsilon\,(a, \mathtt{true}) \qquad {}^{w}\mathtt{write}\, m\, a \triangleq {}^{c}\mathtt{write}\, m \,{}^{c}\!\gg\!= \lambda\_, {}^{c}\varepsilon\,(a, \mathtt{false})$$

Figure 7.9: Alarm writer monad and its specific operators

implementation of ${}^{w}\!\rightsquigarrow$ means that the formal correctness of abstract computations with at least one alarm holds trivially. Hence, on a $\dagger\mathbb{K}$ diagram, an abstract computation fails (i.e. produces no result) as soon as it produces an alarm. On the contrary, in the actual implementation, it produces a result that may be used to find more alarms (without formal guarantee on their meaning).

Figure 7.9 also defines operator $\mathtt{lift}_A : {}^{c}??A \to {}^{w}??A$. Using `lift`, it is straightforward to lift {VPL} abstract domains with computations in the core monad to abstract domains with computations in the alarm writer monad. At last, operator ${}^{w}\mathtt{write}_A : \mathtt{alarm} \to A \to {}^{w}??A$ encodes that ${}^{w}\mathtt{write}\, m\, a$ writes alarm m and returns value a. It is invoked in the implementation of $\dagger\mathbb{K}$ commands that may *fail*: assert (operator "$\vdash$.") and loop (operator ".*").

For example, let us assume here that function $^\sharp\!\dashv\!\neg c : {}^\sharp D \to {}^{c}??{}^\sharp D$ and function $^\sharp\!\sqsubseteq : {}^\sharp D \to {}^\sharp D \to {}^{c}??\mathtt{bool}$ are {VPL} operators from the *core monad* corresponding to those of Figure 7.2. Operator $^\sharp\!\vdash c$, described in Figure 7.3, is implemented in the *alarm writer monad* by the function of type ${}^\sharp D \to {}^{w}??{}^\sharp D$ given below:

$$\begin{array}{ll} \lambda {}^\sharp d, & (\mathtt{lift}\ (^\sharp\!\dashv\!\neg c\ {}^\sharp d))\ {}^{w}\!\gg\!= \lambda {}^\sharp d', \\ & (\mathtt{lift}\ (^\sharp\!\sqsubseteq\ {}^\sharp d'\ {}^\sharp\!\bot))\ {}^{w}\!\gg\!= \lambda b, \\ & \textbf{if}\ b\ \textbf{then}\ {}^{w}\varepsilon\, {}^\sharp d\ \textbf{else}\ ({}^{w}\mathtt{write}\ \texttt{"}\mathit{assert\ \ failure}\texttt{"}\ {}^\sharp d) \end{array}$$

In order to prove that operator $^\sharp\!\vdash c$ is correct w.r.t. its specification $\vdash c$, it suffices to prove the property "$^\sharp\!\vdash c\ {}^\sharp d \rightsquigarrow {}^\sharp d' \Rightarrow \gamma({}^\sharp d) \subseteq [\![c]\!] \wedge {}^\sharp d = {}^\sharp d'$". The proof that this property implies a correct abstraction of $\vdash c$ is independent of the underlying monad.

In summary, the alarm writer monad instantiates our notion of analyzer correctness into "*if the analyzer terminates without raising any alarm*[11]*, then the analyzed program has no runtime error*". Thanks to our compositional design through monads, reasonings on alarm handling appear only in the implementation of the alarm writer monad. Indeed, "raising an alarm" is logically equivalent to a computation that never returns. Actually, Verasco also manages alarms through a writer monad [Jou16]. We have just shown that this feature is very easy to deal with in our framework.

7.4.2 Representation of Concrete Computations

We consider the issue of mechanizing refinement proofs of $\mathbb{K}$ computations. Definition of $\mathbb{K}$ in Section 7.2.1 uses operators inspired from regular expressions. Formally, $\mathbb{K}$ embeds the Kleene algebra[12] of $\mathcal{R}(D, D)$: if K_1 and K_2 are in $\mathcal{R}(D, D)$, then $K_1 ; K_2 = K_1 \cdot K_2$. However, $\mathbb{K}$ does not satisfy itself all properties of a Kleene algebra. In particular, " ;" has two distinct left-zeros $\bot$ and $\lightning$. Thus, it has no right-zero. This forbids applying directly existing Coq tactics for Kleene algebras[BP12].

[11] Formally, the status "no alarm is raised" is given by the Boolean of our alarm writer monad

[12] A Kleene algebra is an idempotent (and thus partially ordered) semiring endowed with a closure operator. It generalizes the operations known from regular expressions: the set of regular expressions over an alphabet is a *free* Kleene algebra.

Like in standard refinement calculus [BvW98], we simplify refinement proofs by computations of weakest-preconditions [Dij75]. More exactly, we use weakest-*liberal*-preconditions (WLP) because they appear naturally in correctness diagrams of abstract computations (illustrated by Figure 7.12 below). [13] Fundamentally, this comes from the fact that weakest-liberal-preconditions do not aim to ensure termination of programs – like our static analyzes – contrary to original weakest-preconditions of Dijkstra.

Definition 7.2 (Weakest-liberal-preconditions)**.** Given $K \in \mathcal{R}(D, D_{\lightning})$, the WLP of K, written here $[K]$, is a function of $\mathcal{P}(D) \to \mathcal{P}(D)$ defined by

$$[K]P \triangleq \{d \in D \mid \forall d' \in D_{\lightning}, d \xrightarrow{K} d' \Rightarrow d' \in P\}$$

To Simplify Refinement Goals by WLP. The main benefit of WLP is to propagate function computations through sequences of relations. Indeed, WLP transforms a sequence into a function composition: $[K_1 ; K_2]P = [K_1]([K_2]P)$. And, given f a function of type $D \to D$, $[\uparrow f]P = \{d \mid f(d) \in P\}$. This allows for instance to compute $[\uparrow f_1 ; \uparrow f_2]P$ as $\{d \mid f_2(f_1(d)) \in P\}$. To understand the benefit of WLP, let us compare this to the direct definition of "$x \xrightarrow{K_1 ; K_2} z$". It induces a formula with an existential quantifier "$\exists y, x \xrightarrow{K_1} y \wedge y \xrightarrow{\uparrow K_2} z$", which, when K_1 is $\uparrow f_1$, can be simplified into a formula without existential quantifier "$f_1(x) \xrightarrow{K_2} z$". In a sense, WLP computations achieve such a simplification for free. Another benefit of WLP is to perform an implicit normalization of computations, in the sense that $[K]P = [\downarrow K]P$ holds.

We embed WLP computations in refinement proofs using the equivalence between $K_1 \sqsubseteq K_2$ and $\forall P, [K_2]P \subseteq [K_1]P$. We list below WLP of main guarded-commands:

$$[\bot]P = D \qquad [\lightning]P = \emptyset \qquad [\varepsilon]P = P$$

$$[\vdash P']P = P' \cap P \qquad [\dashv P']P = (D \setminus P') \cup P$$

$$\left[\bigsqcup_{a \in A} K_a\right]P = \bigcap_{a \in A} [K_a]P \qquad \left[\bigsqcap_{a \in A} K_a\right]P = \bigcup_{a \in A} [K_a]P$$

The methodology of stepwise refinement relies on the fact that $K_1 \sqsubseteq K_2$ implies $K_1 ; K \sqsubseteq K_2 ; K$ and $K ; K_1 \sqsubseteq K ; K_2$. While trying to prove these two properties only from WLP properties above, we observe that the first one derives from $\forall P, [K_2]([K]P) \subseteq [K_1]([K]P)$, itself implied by $K_1 \sqsubseteq K_2$. However, in order to prove the second one, we need to establish an additional property on $[K]$: it is a *monotone predicate transformer*. This means that if $P_1 \subseteq P_2$ then $[K]P_1 \subseteq [K]P_2$.

A Shallow Embedding of WLP Computations. In the style of [#Bou07], we use a shallow embedding of WLP computations, meaning that we avoid the introduction of abstract syntax trees for $\mathbb{K}$ computations, which would induce many difficulties because of binders in $\bigsqcup$ and $\bigsqcap$ operators. Instead, we represent $\mathbb{K}$ computations directly as monotone predicate transformers. In other words, our syntax for $\mathbb{K}$ guarded commands is directly provided by a given set of Coq operators on monotone predicate transformers (corresponding to some WLP computations).

Actually, by exploiting type isomorphism $\mathcal{P}(D) \to \mathcal{P}(D) \simeq D \to \mathcal{P}(\mathcal{P}(D))$, we encode monotone predicate transformers as functions $D \to \mathbb{P}(D)$ where $\mathbb{P}$ is the monad of *monotone predicates of*

[13] These WLP-computations on $\mathbb{K}$ (the concrete semantics) must not be confused with those performed by the `wlp_simplify` tactic, presented in Section 2.2.1. The latter will only be used in Section 7.4.3 to simplify some proofs on the abstract semantics $\sharp\mathbb{K}$.

```
Record ℙ(A:Type) :={
  app:> (A → Prop) → Prop;
  app_monot(P Q:A → Prop): app P → (∀ d, P d → Q d) → app Q}.
```

$$k_1 \,{}^{\mathbb{P}}\!\sqsubseteq k_2 \triangleq \forall P, (k_2\ P) \to (k_1\ P)$$

$${}^{\mathbb{P}}\varepsilon\ a \triangleq \{\mathsf{app} := \lambda P, (P\ a)\} \qquad k_1 \,{}^{\mathbb{P}}\!\gg\!= k_2 \triangleq \{\mathsf{app} := \lambda P, (k_1\ \lambda a, (k_2\ a\ P))\}$$

$${}^{\mathbb{P}}\vdash P' \triangleq \{\mathsf{app} := \lambda P, P' \wedge (P\ \mathtt{tt})\} \qquad {}^{\mathbb{P}}\dashv P' \triangleq \{\mathsf{app} := \lambda P, P' \to (P\ \mathtt{tt})\}$$

$$\overset{A}{\sqcup} \triangleq \{\mathsf{app} := \lambda P, \forall a{:}A, (P\ a)\} \qquad \overset{A}{\sqcap} \triangleq \{\mathsf{app} := \lambda P, \exists a{:}A, (P\ a)\}$$

Figure 7.10: Coq definitions for main operators of monad $\mathbb{P}$

$$\mathbb{K} \triangleq D \to \mathbb{P}\ D \qquad K_1 \sqsubseteq K_2 \triangleq \forall d, (K_1\ d)\ {}^{\mathbb{P}}\!\sqsubseteq (K_2\ d)$$

$$\uparrow f \triangleq \lambda d, {}^{\mathbb{P}}\varepsilon\ (f\ d) \qquad K_1\,;K_2 \triangleq \lambda d, (K_1\ d)\ {}^{\mathbb{P}}\!\gg\!= K_2$$

$$\vdash P' \triangleq \lambda d, {}^{\mathbb{P}}\vdash (P'\ d)\ {}^{\mathbb{P}}\!\gg\!= \lambda\_, ({}^{\mathbb{P}}\varepsilon\ d) \qquad \dashv P' \triangleq \lambda d, {}^{\mathbb{P}}\dashv (P'\ d)\ {}^{\mathbb{P}}\!\gg\!= \lambda\_, ({}^{\mathbb{P}}\varepsilon\ d)$$

$$\Big\lfloor_{a:A} K_a \triangleq \lambda d, \overset{A}{\sqcup}\ {}^{\mathbb{P}}\!\gg\!= \lambda a{:}A, (K_a\ d) \qquad \Big\lceil_{a:A} K_a \triangleq \lambda d, \overset{A}{\sqcap}\ {}^{\mathbb{P}}\!\gg\!= \lambda a{:}A, (K_a\ d)$$

Figure 7.11: Coq definitions for main $\mathbb{K}$ operators

predicate (that we define below). Indeed, they are simpler and more general than monotone predicate transformers: all composition operators of predicate transformers can be derived by combining only *atomic* operators with the $\gg\!=$ operator of monad $\mathbb{P}$. For instance, in Figure 7.11, the A-indexed meet operator of $\mathbb{K}$ is derived from the atomic operator $\overset{A}{\sqcap}$ of $\mathbb{P}$.

Figure 7.10 sketches the Coq definitions of monad $\mathbb{P}$. An element of type $(\mathbb{P}\ A)$ is a record with two fields: a field `app` representing a predicate of $\mathcal{P}(\mathcal{P}(A))$, and a field `app_monot` that is a proof that `app` is monotone. Here, elements of $(\mathbb{P}\ A)$ are implicitly coerced into functions through field `app`. In Figure 7.10, each record definition generates a proof obligation for the missing field `app_monot`. Assert (resp. assume) operator of $\mathbb{P}$ monad is written ${}^{\mathbb{P}}\vdash P'$ (resp. ${}^{\mathbb{P}}\dashv P'$) where P' is of type `Prop`. These operators are of type "$\mathbb{P}$ `unit`" where `unit` is a singleton type which inhabitant is `tt`. Operators $\overset{A}{\sqcap}$ and $\overset{A}{\sqcup}$ are of type $\mathbb{P}\ A$.

A Lightweight Formalization of $\mathbb{K}$ in Coq. Figure 7.11 illustrates how we derive guarded-commands of $\mathbb{K}$ from operators of $\mathbb{P}$ monad. With this representation change, a relation Q in $\mathcal{R}(D, D)$ is now embedded in $\mathbb{K}$ as

$$\overline{Q} \triangleq \Big\lfloor\!\Big\rfloor_{d' \in D} \dashv \{d \mid d \overset{Q}{\mapsto} d'\}\,; d'$$

We can thus still express Hoare specifications (P, Q) of $\mathcal{P}(D) \times \mathcal{R}(D, D)$ by $\vdash P; \overline{Q}$. Hence, we express unbounded iteration by a meet over inductive invariants as explained in Section 7.2.1.

In contrast to [#Bou07], we have not proved in Coq the properties of $\mathbb{K}$ algebra. On refinement goals, we let Coq compute weakest-preconditions and simply solve the remaining goal with standard

```
Record ♯𝕂: Type :={
  impl:♯D →??♯D; spec:𝕂;
  impl_correct: ∀♯d, WHEN (impl ♯d) ⇝♯d' THEN ∀d, d ∈ γ(♯d) → (spec d γ(♯d')) }.
```

Figure 7.12: Sketch of the Coq definition for $\sharp\mathbb{K}$ datatype

Coq tactics. This gives us well-automated proof scripts in practice. Thus, Coq code for $\mathbb{K}$ operators (with $\mathbb{P}$ included) remains very small (around 150 lines, proofs and comments included).

7.4.3 Representations of Correctness Diagrams

The Coq definition of $\sharp\mathbb{K}$ datatype, sketched in Figure 7.12, is actually parametrized by a structure of may-return monad: abstract computations are functions of $\sharp D \to ??\sharp D$. Here, $\sharp D$ equipped with its operators (satisfying the interface given at Figure 7.2) is also a parameter of the definition. Thus, our modular design allows to have abstract computations that do handle alarms, like in our toy analyzer, or that do not, like in our linearization procedure. Indeed, in abstract interpreters, detection of runtime errors (and handling of alarm) is generally done at the top-level interpreter of the analyzer, but not in the internal levels. Our notion of diagram can handle both cases in a generic way.

Therefore, Figure 7.12 defines values of $\sharp\mathbb{K}$ as triples with a field `impl` being an abstract computation, a field `spec` being a concrete computation and a field `impl_correct` being a proof that `impl` is correct w.r.t `spec`. Such proofs are simplified by applying together the WLP embedded in `spec` and the `wlp_simplify` tactic of Section 2.2.1. The latter indeed simplifies reasonings with $\rightsquigarrow$ relation. At last, `impl` being the only informative[14] field of $\sharp\mathbb{K}$ record, type $\sharp\mathbb{K}$ is extracted as OCaml type $\sharp D \to \sharp D$. Similarly, a $\sharp\mathbb{K}$ command is extracted exactly as its underlying abstract computation. Here again, the Coq code for $\sharp\mathbb{K}$ operators (diagrammatic proofs included) is small (around 200 lines, without the implementation of the alarm writer monad).

7.5 Conclusion & Perspectives

We extended the {VPL} with certified handling of non-linear multiplications by a modular and novel design. Our computations are performed by an untrusted oracle delivering a certificate to a certified front-end. Our proofs use diagrammatic constructs based on stepwise refinement calculus. Refinement proofs are finally made clear and concise by the computations of Weakest-Liberal-Preconditions.

Our linearization procedure is able to give a fast over-approximation of integer polynomials thanks to variable intervalization. The precision is increased by domain partitioning (implicitly done with a Continuation-Passing-Style design) and the dynamic computation of bounding affine terms, enabling to finely tune the precision-versus-efficiency trade-off in the oracle.

Because floating-point arithmetic requires to explicitly handle error terms at each operation, {VPL} does not currently support floating-points variables, and our linearization neither. Most non-linear arithmetic used in real-life programs involve floating-points. Therefore, it is hard to evaluate our method on real-life programs. Hence, our experiments are limited to small handmade examples inspired by polynomials often encountered in real-life code, such as parabola or barycentres. On these cases, our oracle is able to give much more precise approximations than the Verasco interval domain.

[14] In Coq jargon, something is "informative" if it is "not a piece of proof" (thus, it remains at extraction).

Our linearization procedure needs also to be extended with others arithmetic operators such as integer division and integer modulo. A simple approach in this direction would: 1) replace each call to these operators by a fresh temporary variable; 2) express the meaning of these operations as nondeterministic assignments of their corresponding variables, using only polynomials, i.e. if t_1 and t_2 are positive then $q := t_1/t_2$ is replaced by $q :\in \{x \mid t_1 - t_2 < x \times t_2 \leq t_1\}$; 3) eliminate temporary variables out of approximated guards. The {VPL} already provides the bricks for such an approach.

At last, we certified a toy analyzer from big-steps semantics of Figure 7.1, by interpreting the operators of concrete semantics in abstract semantics, according to the correspondence of Figure 7.3. We detailed how this toy analyzer handles alarms in the style of Verasco. This could give some hints to adapt Jourdan's framework for Verasco [Jou16] with impure operators on abstract domains and some dynamic strategies of trace partitioning.

Chapter 8

Scientific Production

Contents

8.1 Formally Verified Software

{CompCert-KVX} The Kalray-Verimag CompCert compiler

Status Open-Source prototype, an extension of the official CompCert[1], developed in 2018-2021.
Distribution https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx
Authors Cyril Six, Sylvain Boulmé, David Monniaux and Léo Gourdin.
Features Optimizes Instruction-Level Parallelism and provides a backend for the Kalray VLIW KVX processor, with Scheduling Optimizations for Aarch64 and Risc-V.
Contribution The first (scaling) formally verified compiler that optimizes pipeline usage. Also the first verified compiler for a VLIW architecture.
Source Code official CompCert + 50Kloc of Coq + 20Kloc of OCaml + a few hundred lines of C and assembly.
Papers [#SBM20] (with badge "Artifacts Evaluated Reusable"[2]) and [Six21; MS21; Gou21].

{Impure} FVDP in Coq+OCaml

Status Open-Source prototype, developed in 2018.
Distribution https://github.com/boulme/ImpureDemo
Author Sylvain Boulmé
Features A library to embed impure OCaml oracles within Coq-verified code (FFI of Coq toward OCaml through extraction).
Contribution A preliminary version of this library is included in the {VPL}. This version makes it a standalone library: it is used in {CompCert-KVX} and {SatAnsCert}.
Source Code 700 lines of Coq et 200 lines of OCaml.

[1] https://compcert.org/
[2] https://dl.acm.org/doi/10.1145/3428197

{VPL} The Verified Polyhedron Library

Status Open-Source prototype, developed in 2012-2018.
Distribution https://github.com/VERIMAG-Polyhedra/VPL
Authors Sylvain Boulmé, Alexis Fouihlé, Alexandre Maréchal, David Monniaux, Michaël Périn et Hang Yu
Features Standard operations on an abstract domain of convex polyhera in unbounded dimensions (similar but not formally verified libraries: PPL[3], Apron[4], Polylib[5]...).
Contribution The first formally verified library of convex polyhedra (and a new algorithmic—found by Maréchal and Périn [MP17]—for convex polyhedra libraries: parametric linear programming).
Source Code 45Kloc of OCaml, 2Kloc of C++ and 15Kloc of Coq.
Papers [#FB14; #Bou+18; #BM19] and [FMP13; Fou15; Mar+16; Mar17; MP17; MMP17].
External Users the Verasco[6] static analyzer (see [Jou+15; Jou16]) and a verified polyhedral compiler [CL21].

{VplTactic} Simplifying Rational Inequalities in Coq proofs

Status Open-Source prototype, developed in 2016-2017.
Distribution https://github.com/VERIMAG-Polyhedra/VplTactic
Authors Sylvain Boulmé and Alexandre Maréchal
Features A Coq tactic (plugin) to simplify rational inequalities in Coq proofs.
Contribution With respect to `omega` or `lia` tactics, this tactic never fails, but discovers instead the equalities deducible from the inequalities (and remove redundant inequalities); this may help other Coq tactics to benefit from these discovered equalities. Returning "UNSAT" or a list of learned equalities is standard in arithmetic procedures of many SMT-solvers (e.g. using a Nelson-Oppen approach with Shostak theories [MZ02]). This feature is here provided in Coq interactive proofs.
Source Code 800 lines of OCaml and 800 lines of Coq (on the top of the {VPL}).
Papers [#BM18] and [Mar17]

{SatAnsCert} Certification of Boolean SAT-solver Answers

Status Open-Source prototype, developed in 2018.
Distribution https://github.com/boulme/satans-cert
Authors Sylvain Boulmé and Thomas Vandendorpe
Features A verified verifier of Boolean SAT-solvers answers (for SAT-solvers at the state-of-the-art in 2018). SAT anwsers are verified from a model retourned by the SAT-solver. UNSAT answers are verified from the DRAT proof retourned by the SAT-solver (with the help of drat-trim[7]).
Contribution UNSAT answers are verified according principles of [Cru+], but {SatAnsCert} provides a much more efficient Coq-verified verifier thanks to an untrusted *Polymorphic LCF style* parser of LRAT proofs programmed in OCaml.
Source Code 1Kloc of Coq et 1Kloc of OCaml (on the top of {Impure}).

[3] http://bugseng.com/products/ppl/
[4] http://apron.cri.ensmp.fr/
[5] https://www.irisa.fr/polylib/
[6] http://compcert.inria.fr/verasco
[7] https://github.com/marijnheule/drat-trim

8.2 Peer-Reviewed Publications

[#Bou07] Sylvain Boulmé. “Intuitionistic Refinement Calculus.” In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Vol. 4583. Lecture Notes in Computer Science. Springer, 2007, pp. 54–69. DOI: 10.1007/978-3-540-73228-0_6.

[#BM15] Sylvain Boulmé and Alexandre Maréchal. “Refinement to Certify Abstract Interpretations, Illustrated on Linearization for Polyhedra.” In: *ITP'15*. Vol. 9236. LCNS. Springer, 2015, pp. 100–116. DOI: 10.1007/978-3-319-22102-1_7.

[#BM18] Sylvain Boulmé and Alexandre Maréchal. “A Coq Tactic for Equality Learning in Linear Arithmetic.” In: *ITP'18*. Vol. 10895. LNCS. Springer, 2018, pp. 108–125. DOI: 10.1007/978-3-319-94821-8_7.

[#BM19] Sylvain Boulmé and Alexandre Maréchal. “Refinement to Certify Abstract Interpretations: Illustrated on Linearization for Polyhedra.” In: *J. Autom. Reasoning* 62.4 (2019), pp. 505–530. DOI: 10.1007/s10817-018-9492-2. URL: https://hal.archives-ouvertes.fr/hal-01133865.

[#Bou+18] Sylvain Boulmé et al. “The Verified Polyhedron Library: an Overview.” In: *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2018, Timisoara, Romania, September 20-23, 2018*. IEEE Computer Society, 2018, pp. 9–17. DOI: 10.1109/SYNASC.2018.00014. URL: https://hal.archives-ouvertes.fr/hal-02100006.

[#FB14] Alexis Fouilhé and Sylvain Boulmé. “A Certifying Frontend for (Sub)polyhedral Abstract Domains.” In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*. Vol. 8471. Lecture Notes in Computer Science. Springer, 2014, pp. 200–215. DOI: 10.1007/978-3-319-12154-3_13. URL: https://hal.archives-ouvertes.fr/hal-00991853.

[#SBM20] Cyril Six, Sylvain Boulmé, and David Monniaux. “Certified and Efficient Instruction Scheduling: Application to Interlocked VLIW Processors.” In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428197. URL: https://hal.archives-ouvertes.fr/hal-02185883.

Bibliography

[AI20] French National Cybersecurity Agency (ANSSI) and Inria. *Requirements on the Use of Coq in the Context of Common Criteria Evaluations*. Sept. 2020. URL: https://www.ssi.gouv.fr/uploads/2014/11/anssi-requirements-on-the-use-of-coq-in-the-context-of-common-criteria-evaluations-v1.0-en.pdf.

[Abr96] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996. ISBN: 978-0-521-02175-3. DOI: 10.1017/CBO9780511624162.

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. "A Stratified Semantics of General References Embeddable in Higher-Order Logic." In: *LICS'02*. Washington, DC, USA: IEEE, 2002, p. 75. DOI: 10.1109/LICS.2002.1029818.

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. "State-dependent representation independence." In: *POPL'09*. ACM, 2009, pp. 340–353. DOI: 10.1145/1480881.1480925.

[AK17] Xavier Allamigeon and Ricardo D. Katz. "A Formalization of Convex Polyhedra Based on the Simplex Method." In: *ITP'17*. Vol. 10499. LNCS. Springer, 2017, pp. 28–45. DOI: 10.1007/978-3-319-66107-0_3.

[AM17] Abhishek Anand and Greg Morrisett. "Revisiting Parametricity: Inductives and Uniformity of Propositions." In: *CoRR* abs/1705.01163 (2017). arXiv: 1705.01163.

[App01] Andrew W. Appel. "Foundational Proof-Carrying Code." In: *LICS'01*. USA: IEEE Computer Society, 2001, p. 247. DOI: 10.5555/871816.871860.

[App14] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN: 978-1-10-704801-0. URL: https://www.cs.princeton.edu/~appel/papers/plcc.pdf.

[App+03] Andrew W. Appel et al. "A Trustworthy Proof Checker." In: *J. Autom. Reason.* 31.3-4 (2003), pp. 231–260. DOI: 10.1023/B:JARS.0000021013.61329.58. URL: https://doi.org/10.1023/B:JARS.0000021013.61329.58.

[App+07] Andrew W. Appel et al. "A Very Modal Model of a Modern, Major, General Type System." In: *POPL'07*. New York, NY, USA: ACM Press, 2007, pp. 109–122. ISBN: 1-59593-575-4.

[Arm+10] Michaël Armand et al. "Extending Coq with Imperative Features and Its Application to SAT Verification." In: *ITP'10*. Vol. 6172. LNCS. Springer, 2010, pp. 83–98. DOI: 10.1007/978-3-642-14052-5_8.

[Arm+11] Michaël Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses." In: *Certified Programs and Proofs (CPP)*. Vol. 7086. LNCS. Springer, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[Bac80] Ralph-Johan Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*. (Amsterdam) Mathematical Centre, 1980. URL: https://tucs.fi/publications/view/?pub_id=bBack_RalphxJohan80a.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. ISBN: 978-0-387-98417-9. DOI: 10.1007/978-1-4612-1674-2.

[BB02] Henk Barendregt and Erik Barendsen. "Autarkic Computations in Formal Proofs." In: *Journal of Automated Reasoning* 28.3 (2002), pp. 321–336.

[Bar99] Bruno Barras. "Auto-validation d'un système de preuves avec familles inductives." Thèse de Doctorat. Université Paris 7, Nov. 1999.

[Bed+12] Ricardo Bedin França et al. "Formally verified optimizing compilation in ACG-based flight control software." In: *ERTS2*. Feb. 2012. HAL: hal-00653367.

[BKM05] Florence Benoy, Andy King, and Frédéric Mesnard. "Computing Convex Hulls with a Linear Solver." In: *Theory and Practice of Logic Programming* 5.1-2 (Jan. 2005).

[BM12] Jean-Philippe Bernardy and Guilhem Moulin. "A Computational Interpretation of Parametricity." In: *LICS'12*. IEEE Computer Society, 2012. DOI: 10.1109/LICS.2012.25.

[BM13] Jean-Philippe Bernardy and Guilhem Moulin. "Type-theory in color." In: *ICFP'13*. ACM Press, 2013. DOI: 10.1145/2500365.2500577.

[Bes06] Frédéric Besson. "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond." In: *Types for Proofs and Programs (TYPES)*. Vol. 4502. LNCS. Springer, 2006, pp. 48–62. DOI: 10.1007/978-3-540-74464-1_4.

[BBW19a] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Verified CompCert Front-End for a Memory Model Supporting Pointer Arithmetic and Uninitialised Data." In: *J. Autom. Reason.* 62.4 (2019), pp. 433–480. DOI: 10.1007/s10817-017-9439-z.

[BBW19b] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics." In: *J. Autom. Reason.* 63.2 (2019), pp. 369–392. DOI: 10.1007/s10817-018-9496-y.

[BCP11] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. "Modular SMT Proofs for Fast Reflexive Checking Inside Coq." In: *Certified Programs and Proofs (CPP)*. Vol. 7086. LNCS. Springer, 2011, pp. 151–166.

[Bes+07] Frédéric Besson et al. *Result certification for relational program analysis*. Research Report RR-6333. INRIA, 2007. HAL: inria-00166930.

[Bes+10] Frédéric Besson et al. "Certified Result Checking for Polyhedral Analysis of Bytecode Programs." In: *Trustworthy Global Computing (TGC)*. Vol. 6084. LNCS. Springer, 2010, pp. 253–267.

[Bir+11] Lars Birkedal et al. "Step-indexed Kripke Models over Recursive Worlds." In: *POPL'11*. Austin, Texas, USA: ACM Press, 2011, pp. 119–132. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926401.

[BRA10] Sandrine Blazy, Benot Robillard, and Andrew W. Appel. "Formal Verification of Coalescing Graph-Coloring Register Allocation." In: *ESOP'10*. Vol. 6012. LNCS. Springer, 2010, pp. 145–164. DOI: 10.1007/978-3-642-11957-6_9.

[Bla+13] Sandrine Blazy et al. “Formal Verification of a C Value Analysis Based on Abstract Interpretation.” In: *SAS'13*. Vol. 7935. LNCS. Springer, 2013.

[BJM14] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. “Implementing and Reasoning About Hash-consed Data Structures in Coq.” In: *J. Autom. Reasoning* 53.3 (2014), pp. 271–304. DOI: 10.1007/s10817-014-9306-0.

[BP12] Thomas Braibant and Damien Pous. “Deciding Kleene Algebras in Coq.” In: *Logical Methods in Computer Science* 8.1 (2012).

[Bre+18] Joachim Breitner et al. “Ready, Set, Verify! Applying Hs-to-coq to Real-world Haskell Code (Experience Report).” In: *Proc. ACM Program. Lang.* 2.ICFP'18 (July 2018), 89:1–89:16. ISSN: 2475-1421. DOI: 10.1145/3236784.

[Car+17] João F. N. Carvalho et al. “The Register Allocation and Instruction Scheduling Challenge.” In: *Proceedings of the 21st Brazilian Symposium on Programming Languages*. SBLP 2017. Fortaleza, CE, Brazil: ACM, 2017, 3:1–3:9. DOI: 10.1145/3125374.3125380. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-232189.

[CSW14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. “Combining proofs and programs in a dependently typed language.” In: *POPL'14*. ACM, 2014, pp. 33–46. DOI: 10.1145/2535838.2535883.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN: 978-0-262-02665-9. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

[Chl+09] Adam Chlipala et al. “Effective interactive proofs for higher-order imperative programs.” In: *ICFP'09*. ACM, 2009, pp. 79–90. DOI: 10.1145/1596550.1596565.

[Chv83] Vasek Chvatal. *Linear Programming*. Series of books in the Mathematical Sciences. W. H. Freeman, 1983. ISBN: 9780716715870.

[Cim+18] Alessandro Cimatti et al. “Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions.” In: *ACM Trans. Comput. Logic* 19.3 (Aug. 2018). ISSN: 1529-3785. DOI: 10.1145/3230639.

[Cla+13] Guillaume Claret et al. “Lightweight proof by reflection using a posteriori simulation of effectful computation.” In: *ITP'13*. Vol. 7998. LNCS. 2013.

[CP88] Thierry Coquand and Christine Paulin. “Inductively defined types.” In: *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*. Vol. 417. Lecture Notes in Computer Science. Springer, 1988, pp. 50–66. DOI: 10.1007/3-540-52335-9_47.

[CL21] Nathanaël Courant and Xavier Leroy. “Verified code generation for the polyhedral model.” In: *Proc. ACM Program. Lang.* 5.POPL'21 (2021), 40:1–40:24. DOI: 10.1145/3434321. URL: https://xavierleroy.org/publi/polyhedral-codegen.pdf.

[Cou02] Patrick Cousot. “Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation.” In: *Theoretical Computer Science* 277.1-2 (2002). DOI: 10.1016/S0304-3975(00)00313-3.

[CC77] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.” In: *POPL'77*. ACM Press, 1977. DOI: 10.1145/512950.512973.

[CH78] Patrick Cousot and Nicolas Halbwachs. “Automatic discovery of linear restraints among variables of a program.” In: *POPL’78*. ACM Press, 1978. DOI: 10.1145/512760.512770.

[CMS17] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. “Efficient Certified Resolution Proof Checking.” In: *Tools and Algorihtms for the Construction and Analysis of Systems (TACAS)*. Vol. 10205. LNCS. Springer, 2017, pp. 118–135. DOI: 10.1007/978-3-662-54577-5_7.

[Cru+] Luís Cruz-Filipe et al. “Efficient Certified RAT Verification.” In: *CADE’17*. URL: https://imada.sdu.dk/~petersk/lrat/paper.pdf.

[dBru68] N.G. de Bruijn. “The Mathematical Language AUTOMATH, Its Usage, and Some of Its Extensions.” In: *Symposium on Automatic Demonstration*. Vol. 125. LNM. Versailles, France: Springer, Dec. 1968, pp. 29–61. URL: http://www.cs.cornell.edu/courses/cs4860/2018fa/lectures/The-mathematical-language-Automath_de-Bruijn.pdf.

[dRE98] Willem P. de Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*. Vol. 46. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998. ISBN: 0-521-64170-5.

[Dij68] Edsger W. Dijkstra. “A Constructive Approach to the Problem of Program Correctness.” In: *BIT Numerical Mathematics* 8 (1968). DOI: 10.1007/BF01933419. URL: https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF.

[Dij72] Edsger W. Dijkstra. “The Humble Programmer.” In: *ACM Turing Award Lectures 2007*. New York, NY, USA: Association for Computing Machinery, 1972. ISBN: 9781450310499. DOI: 10.1145/1283920.1283927.

[Dij75] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs.” In: *Commun. ACM* 18.8 (1975), pp. 453–457. DOI: 10.1145/360933.360975.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: https://seriouscomputerist.atariverse.com/media/pdf/book/Discipline%20of%20Programming.pdf.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. “Logical Step-Indexed Logical Relations.” In: *LICS’09*. IEEE Computer Society, 2009, pp. 71–80. DOI: 10.1109/LICS.2009.34.

[Eki+17] Burak Ekici et al. “SMTCoq: A Plug-In for Integrating SMT Solvers into Coq.” In: *CAV’17*. Vol. 10427. LNCS. Springer, 2017, pp. 126–133. DOI: 10.1007/978-3-319-63390-9_7.

[Far02] Julius Farkas. “Theorie der einfachen Ungleichungen.” In: *Journal für die Reine und Angewandte Mathematik* 124 (1902).

[Far12] Rida T. Farouki. “The Bernstein polynomial basis: A centennial retrospective.” In: *Computer Aided Geometric Design* 29.6 (2012). URL: http://www.sciencedirect.com/science/article/pii/S0167839612000192 (visited on 05/23/2014).

[Fer+01] Christian Ferdinand et al. “Reliable and Precise WCET Determination for a Real-Life Processor.” In: *EMSOFT’01*. Vol. 2211. LNCS. Springer, 2001, pp. 469–485. DOI: 10.1007/3-540-45449-7_32.

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. “Type-safe modular hash-consing.” In: *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. ACM, 2006, pp. 12–19. DOI: 10.1145/1159876.1159880.

[Fou15] Alexis Fouilhé. “Revisiting the abstract domain of polyhedra: constraints-only representation and formal proof.” PhD thesis. Université Grenoble Alpes, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01286086.

[FMP13] Alexis Fouilhé, David Monniaux, and Michaël Périn. “Efficient Generation of Correctness Certificates for the Abstract Domain of Polyhedra.” In: *SAS'13*. Vol. 7935. LNCS. Springer, 2013, pp. 345–365. DOI: 10.1007/978-3-642-38856-9_19.

[Fou27] Joseph Fourier. “Histoire de l'Académie, partie mathématique (1824).” In: *Mémoires de l'Académie des sciences de l'Institut de France* 7 (1827).

[Fra+11] Ricardo Bedin França et al. “Towards Formally Verified Optimizing Compilation in Flight Control Software.” In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France*. Vol. 18. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011, pp. 59–68. DOI: 10.4230/OASIcs.PPES.2011.59.

[GM93] Paul H. B. Gardiner and Carroll Morgan. “A Single Complete Rule for Data Refinement.” In: *Formal Aspects Comput.* 5.4 (1993), pp. 367–382. DOI: 10.1007/BF01212407.

[Gar02] Jacques Garrigue. “Relaxing the Value Restriction.” In: *Asian Programming Languages and Systems Symposium (APLAS)*. Vol. 2998. LNCS. Springer, 2002, pp. 31–45.

[Gel08] Allen Van Gelder. “Verifying RUP Proofs of Propositional Unsatisfiability.” In: *International Symposium on Artificial Intelligence and Mathematics*. 2008.

[Gen13] Ian P. Gent. “Optimal Implementation of Watched Literals and More General Techniques.” In: 48 (2013), pp. 231–251. DOI: 10.1613/jair.4016.

[GW14] Jeremy Gibbons and Nicolas Wu. “Folding domain-specific languages: deep and shallow embeddings (functional Pearl).” In: *ICFP'14*. ACM, 2014, pp. 339–347. DOI: 10.1145/2628136.2628138.

[Göd31] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I.” In: *Monatshefte für Mathematik und Physik* (1931). DOI: 10.1007/BF01700692.

[GN03] Evguenii I. Goldberg and Yakov Novikov. “Verification of Proofs of Unsatisfiability for CNF Formulas.” In: *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 2003, pp. 10886–10891. DOI: 10.1109/DATE.2003.10008.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Vol. 78. LNCS. Springer, 1979. DOI: 10.1007/3-540-09724-4.

[Gor+78] Michael J. C. Gordon et al. “A Metalanguage for Interactive Proof in LCF.” In: *POPL'78*. ACM Press, 1978, pp. 119–130. DOI: 10.1145/512760.512773.

[Gou21] Léo Gourdin. “PhD Student session: formally verified postpass scheduling with peephole optimization for AArch64.” In: *20èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL 2021*. June 2021. URL: https://www.lirmm.fr/afadl2021/papers/afadl2021_paper_9.pdf.

[GM05] Benjamin Grégoire and Assia Mahboubi. “Proving Equalities in a Commutative Ring Done Right in Coq.” In: *TPHOL*. 2005, pp. 98–113. DOI: 10.1007/11541868_7.

[GMZ00] Dan Grossman, Greg Morrisett, and Steve Zdancewic. “Syntactic Type Abstraction.” In: *ACM Trans. Program. Lang. Syst.* 22.6 (2000), pp. 1037–1080. DOI: 10.1145/371880.371887.

[Han88] David Handelman. “Representing polynomials by positive linear functions on compact convex polyhedra.” In: *Pacific Journal of Mathematics* 132.1 (1988).

[HT98] John Harrison and Laurent Théry. “A Skeptic’s Approach to Combining HOL and Maple.” In: *Journal of Automated Reasoning* 21.3 (1998), pp. 279–294. DOI: 10.1023/A:1006023127567. URL: https://www.cl.cam.ac.uk/~jrh13/papers/cas.html.

[Heu16] Marijn Heule. “The DRAT format and DRAT-trim checker.” In: *CoRR* abs/1610.06229 (2016). arXiv: 1610.06229. URL: http://arxiv.org/abs/1610.06229.

[HJW13a] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. “Trimming while checking clausal proofs.” In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 181–188. URL: http://ieeexplore.ieee.org/document/6679408/.

[HJW13b] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. “Verifying Refutations with Extended Resolution.” In: *CADE’13*. Vol. 7898. LNCS. Springer, 2013, pp. 345–359. DOI: 10.1007/978-3-642-38574-2_24.

[HJW14] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. “Bridging the gap between easy generation and efficient verification of unsatisfiability proofs.” In: *Softw. Test., Verif. Reliab.* 24.8 (2014), pp. 593–607. DOI: 10.1002/stvr.1549.

[HKB17] Marijn Heule, Benjamin Kiesl, and Armin Biere. “Short Proofs Without New Variables.” In: *CADE’17*. Vol. 10395. LNCS. Springer, 2017, pp. 130–147. DOI: 10.1007/978-3-319-63046-5_9.

[Hoa72] C. A. R. Hoare. “Proof of Correctness of Data Representations.” In: *Acta Informatica* 1 (1972), pp. 271–281. DOI: 10.1007/BF00289507.

[HDA10] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. “A Theory of Indirection via Approximation.” In: *POPL’10*. Madrid, Spain: ACM Press, 2010, pp. 171–184. DOI: 10.1145/1706299.1706322.

[HK12] Jacob M. Howe and Andy King. “Polyhedral Analysis using Parametric Objectives.” In: *Static Analysis Symposium (SAS)*. Vol. 7460. LNCS. Springer, 2012, pp. 41–57.

[Hwu+93] Wen-mei W. Hwu et al. “The superblock: An effective technique for VLIW and superscalar compilation.” In: *J. Supercomput.* 7.1-2 (1993), pp. 229–248. DOI: 10.1007/BF01205185.

[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.

[JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. “Validating LR(1) parsers.” In: *ESOP 2012: Programming Languages and Systems, 21st European Symposium on Programming*. LNCS 7211. Springer, 2012, pp. 397–416. DOI: 10.1007/978-3-642-28869-2_20.

[Jou+15] Jacques-Herni Jourdan et al. “A Formally-Verified C Static Analyzer.” In: *POPL’15*. Mumbai, India: ACM Press, 2015, pp. 247–259.

[Käs+18] Daniel Kästner et al. “CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler.” In: *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE. Toulouse, France, Jan. 2018, pp. 1–9. URL: https://hal.inria.fr/hal-01643290.

[Kel13] Chantal Keller. “A Matter of Trust: Skeptical Communication Between Coq and External Provers. (Question de confiance : communication sceptique entre Coq et des prouveurs externes).” PhD thesis. École Polytechnique, Palaiseau, France, 2013. URL: https://tel.archives-ouvertes.fr/pastel-00838322.

[KL12] Chantal Keller and Marc Lasson. “Parametricity in an Impredicative Sort.” In: *Computer Science Logic (CSL’12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. Vol. 16. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 381–395. DOI: 10.4230/LIPIcs.CSL.2012.381.

[Kie+20] Benjamin Kiesl et al. “Simulating Strong Practical Proof Systems with Extended Resolution.” In: *J. Autom. Reason.* 64.7 (2020), pp. 1247–1267. DOI: 10.1007/s10817-020-09554-z. URL: https://doi.org/10.1007/s10817-020-09554-z.

[Kin76] James C. King. “Symbolic Execution and Program Testing.” In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[Kin+08] Samuel T. King et al. “Designing and Implementing Malicious Hardware.” In: *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*. LEET’08. San Francisco, California: USENIX Association, 2008.

[Kir+15] Florent Kirchner et al. “Frama-C: A software analysis perspective.” In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. “Formal C semantics: CompCert and the C standard.” In: *ITP 2014: Interactive Theorem Proving*. LNCS 8558. Springer, 2014, pp. 543–548. DOI: 10.1007/978-3-319-08970-6_36.

[KW15] Robbert Krebbers and Freek Wiedijk. “A Typed C11 Semantics for Interactive Theorem Proving.” In: *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*. ACM, 2015, pp. 15–27. DOI: 10.1145/2676724.2693571. URL: http://robbertkrebbers.nl/research/articles/interpreter.pdf.

[Lam17a] Peter Lammich. “Efficient Verified (UN)SAT Certificate Checking.” In: *CADE’17*. Vol. 10395. LNCS. Springer, 2017, pp. 237–254. DOI: 10.1007/978-3-319-63046-5_15.

[Lam17b] Peter Lammich. “The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees.” In: *Theory and Applications of Satisfiability Testing - SAT 2017*. Vol. 10491. LNCS. Springer, 2017, pp. 457–463. DOI: 10.1007/978-3-319-66263-3_29.

[Lam+91] Butler W. Lampson et al. “Authentication in Distributed Systems: Theory and Practice.” In: *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*. ACM, 1991, pp. 165–182. DOI: 10.1145/121132.121160.

[Lap15] Vincent Laporte. “Verified static analyzes for low-level languages.” Theses. Université Rennes 1, Nov. 2015. URL: https://tel.archives-ouvertes.fr/tel-01285624.

[Ler09a] Xavier Leroy. “A formally verified compiler back-end.” In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446. URL: http://xavierleroy.org/publi/compcert-backend.pdf.

[Ler09b] Xavier Leroy. “Formal verification of a realistic compiler.” In: *Communications of the ACM* 52.7 (2009). DOI: 10.1145/1538788.1538814. HAL: inria-00415861.

[Ler11] Xavier Leroy. “Verified squared: does critical software deserve verified tools?” In: *POPL'11*. Austin, TX, USA: ACM, Jan. 2011, pp. 1–2. DOI: 10.1145/1926385.1926387.

[Ler17] Xavier Leroy. *How I found a crash bug with hyperthreading in Intels Skylake processors*. July 2017. URL: https://thenextweb.com/contributors/2017/07/05/found-crash-bug-hyperthreading-intels-skylake-processors/.

[LB08] Xavier Leroy and Sandrine Blazy. “Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations.” In: *J. Autom. Reason.* 41.1 (2008), pp. 1–31. DOI: 10.1007/s10817-008-9099-0.

[Ler+14] Xavier Leroy et al. “The CompCert memory model.” In: *Program Logics for Certified Compilers*. Ed. by Andrew W. Appel. Cambridge University Press, Mar. 2014, pp. 237–271.

[Ler+20] Xavier Leroy et al. *The OCaml System*. INRIA. 2013-2020.

[LR20] Thomas Letan and Yann Régis-Gianas. “FreeSpec: Specifying, Verifying and Executing Impure Computations in Coq.” In: *CPP 2020 - 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Nouvelle-Orléans, United States: ACM, Jan. 2020, pp. 1–15. DOI: 10.1145/3372885.3373812. URL: https://hal.inria.fr/hal-02422273.

[Let04] Pierre Letouzey. “Certified functional programming, Program extraction within Coq proof assistant.” PhD thesis. Université de Paris XI Orsay, 2004.

[Let08] Pierre Letouzey. “Extraction in Coq: An Overview.” In: *Computability in Europe (CiE)*. Vol. 5028. LNCS. Springer, 2008, pp. 359–369. DOI: 10.1007/978-3-540-69407-6_39.

[LH96] Sheng Liang and Paul Hudak. “Modular Denotational Semantics for Compiler Construction.” In: *ESOP'96*. Vol. 1058. Springer, 1996, pp. 219–234.

[Mai+19] Claire Maiza et al. “A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems.” In: *ACM Comput. Surv.* 52.3 (2019), 56:1–56:38. DOI: 10.1145/3323212.

[MZ02] Zohar Manna and Calogero G. Zarba. “Combining Decision Procedures.” In: *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*. Vol. 2757. Lecture Notes in Computer Science. Springer, 2002, pp. 381–422. DOI: 10.1007/978-3-540-40007-3_24.

[Mar17] Alexandre Maréchal. “New Algorithmics for Polyhedral Calculus via Parametric Linear Programming.” PhD thesis. Université Grenoble Alpes, France, 2017. URL: https://tel.archives-ouvertes.fr/tel-01695086.

[MMP17] Alexandre Maréchal, David Monniaux, and Michaël Périn. “Scalable Minimizing-Operators on Polyhedra via Parametric Linear Programming.” In: *Static Analysis Symposium (SAS)*. Vol. 10422. LNCS. 2017. URL: https://hal.archives-ouvertes.fr/hal-01555998.

[MP17] Alexandre Maréchal and Michaël Périn. “Efficient Elimination of Redundancies in Polyhedra by Raytracing.” In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. LNCS. Springer, 2017, pp. 367–385.

[Mar+16] Alexandre Maréchal et al. “Polyhedral Approximation of Multivariate Polynomials Using Handelman’s Theorem.” In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. LNCS. Springer, 2016, pp. 166–184.

[Mat08] Aditya P. Mathur. *Foundations of Software Testing*. 1st. Addison-Wesley Professional, 2008. ISBN: 8131716600.

[MR05] Laurent Mauborgne and Xavier Rival. “Trace Partitioning in Abstract Interpretation Based Static Analyzers.” In: *ESOP’05*. Vol. 3444. LNCS. 2005.

[Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994. ISBN: 0-07-016333-2.

[Mil71] Robin Milner. “An Algebraic Definition of Simulation Between Programs.” In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*. William Kaufmann, 1971, pp. 481–489. URL: http://ijcai.org/Proceedings/71/Papers/044.pdf.

[Mil79] Robin Milner. “LCF: A Way of Doing Proofs with a Machine.” In: *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*. Vol. 74. Lecture Notes in Computer Science. Springer, 1979, pp. 146–159. DOI: 10.1007/3-540-09526-8_11.

[Min06] Antoine Miné. “Symbolic methods to enhance the precision of numerical abstract domains.” In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Vol. 3855. LNCS. Springer, 2006. HAL: hal-00136661.

[Mon03] Jean-François Monin. *Understanding formal methods*. Springer, 2003. URL: https://dl.acm.org/doi/book/10.5555/640600.

[MS21] David Monniaux and Cyril Six. “Simple, Light, yet Formally Verified, Global Common Subexpression Elimination and Loop-Invariant Code Motion.” In: *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2021. ACM, 2021, pp. 85–96. DOI: 10.1145/3461648.3463850.

[Mor94] Carroll Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice Hall, 1994. ISBN: 978-0-13-123274-7.

[MMS15] Mariano M. Moscato, César A. Muñoz, and Andrew P. Smith. “Affine Arithmetic and Applications to Real-Number Proving.” In: *ITP’15*. Vol. 9236. LNCS. Springer, 2015.

[Nan+08] Aleksandar Nanevski et al. “Ynot: dependent types for imperative programs.” In: *ICFP’08*. ACM, 2008, pp. 229–240. DOI: 10.1145/1411204.1411237. URL: https://doi.org/10.1145/1411204.1411237.

[Nec97] George C. Necula. “Proof-Carrying Code.” In: *POPL’97*. ACM Press, 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[Nec00] George C. Necula. “Translation validation for an optimizing compiler.” In: *PLDI'00*. ACM Press, 2000, pp. 83–94. DOI: 10.1145/349299.349314.

[NL96] George C. Necula and Peter Lee. “Safe Kernel Extensions Without Run-Time Checking.” In: *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996*. ACM, 1996, pp. 229–243. DOI: 10.1145/238721.238781.

[NL98] George C. Necula and Peter Lee. “The Design and Implementation of a Certifying Compiler.” In: *PLDI'98*. ACM, 1998, pp. 333–344. DOI: 10.1145/277650.277752.

[Oli+14] André Oliveira Maroneze et al. “A Formally Verified WCET Estimation Tool.” In: *14th International Workshop on Worst-Case Execution Time Analysis*. Madrid, Spain, July 2014. DOI: 10.4230/OASIcs.WCET.2014.11. URL: https://people.irisa.fr/David.Pichardie/papers/wcet14.pdf.

[RC18] Adrian Rebola-Pardo and Luís Cruz-Filipe. “Complete and Efficient DRAT Proof Checking.” In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. IEEE, 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8602993.

[Reg19] Yann Regis-Gianas. “About some Metamorphoses of Computer Programs.” Habilitation à diriger des recherches. Université Paris Diderot, Nov. 2019. URL: https://hal.inria.fr/tel-02405839.

[Rey83] John C. Reynolds. “Types, Abstraction and Parametric Polymorphism.” In: *IFIP Congress*. 1983, pp. 513–523.

[Rey93] John C. Reynolds. “The Discoveries of Continuations.” In: *Lisp and Symbolic Computation* 6.3-4 (1993).

[RL10] Silvain Rideau and Xavier Leroy. “Validating register allocation and spilling.” In: *Compiler Construction (CC 2010)*. Vol. 6011. LNCS. Springer, 2010, pp. 224–243.

[Sch+18] Bernhard Schommer et al. “Embedded Program Annotations for WCET Analysis.” In: *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*. Vol. 63. OASIcs. Dagstuhl Publishing, July 2018. DOI: 10.4230/OASIcs.WCET.2018.8.

[SLM09] João P. Marques Silva, Inês Lynce, and Sharad Malik. “Conflict-Driven Clause Learning SAT Solvers.” In: *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 131–153. DOI: 10.3233/978-1-58603-929-5-131.

[Six21] Cyril Six. “Optimized and formally-verified compilation for a VLIW processor.” PhD thesis. Université Grenoble Alpes, France, July 2021. URL: https://hal.archives-ouvertes.fr/tel-03326923.

[Soz+20] Matthieu Sozeau et al. “Coq Coq correct! verification of type checking and erasure for Coq, in Coq.” In: *Proc. ACM Program. Lang.* 4.POPL'20 (2020), 8:1–8:28. DOI: 10.1145/3371076.

[Spi13] Arnaud Spiwack. “Abstract interpretation as anti-refinement.” In: *CoRR* abs/1310.4283 (2013). URL: http://arxiv.org/abs/1310.4283.

[Swi08] Wouter Swierstra. “Data Types à La Carte.” In: *J. Funct. Program.* 18.4 (July 2008), pp. 423–436. ISSN: 0956-7968. DOI: 10.1017/S0956796808006758. URL: https://doi.org/10.1017/S0956796808006758.

[Tea20] The Coq Development Team. *The Coq Proof Assistant*. 1999-2020. URL: https://coq.inria.fr/.

[Tho84] Ken Thompson. "Reflections on Trusting Trust." In: *Commun. ACM* 27.8 (Aug. 1984), pp. 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. "Evaluating value-graph translation validation for LLVM." In: *PLDI'11*. ACM, 2011, pp. 295–305. DOI: 10.1145/1993498.1993533. URL: https://dash.harvard.edu/bitstream/handle/1/4762396/pldi84-tristan.pdf.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. "Formal Verification of Translation Validators: a Case Study on Instruction Scheduling Optimizations." In: *POPL'08*. ACM Press, 2008, pp. 17–27. DOI: 10.1145/1328438.1328444.

[VW07] Dimitrios Vytiniotis and Stephanie Weirich. "Free Theorems and Runtime Type Representations." In: *Electronic Notes in Theoretical Computer Science* 173 (2007), pp. 357–373. DOI: 10.1016/j.entcs.2007.02.043.

[Wad89] Philip Wadler. "Theorems for Free!" In: *Functional Programming Languages and Computer Architecture (FPCA)*. ACM Press, 1989, pp. 347–359. DOI: 10.1145/99370.99404.

[Wad95] Philip Wadler. "Monads for Functional Programming." In: *Advanced Functional Programming*. Vol. 925. LNCS. Springer, 1995.

[WSS13] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. "FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis." In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, pp. 697–708. ISBN: 9781450324779. DOI: 10.1145/2508859.2516654.

[WWS19] Yuting Wang, Pierre Wilke, and Zhong Shao. "An abstract stack based approach to verified compositional compilation to machine code." In: *Proc. ACM Program. Lang.* 3.POPL'19 (2019), 62:1–62:30. DOI: 10.1145/3290375.

[Wan+20] Yuting Wang et al. "CompCertELF: verified separate compilation of C programs into ELF object files." In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 197:1–197:28. DOI: 10.1145/3428265. URL: https://doi.org/10.1145/3428265.

[WHJ13] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. "Mechanical Verification of SAT Refutations with Extended Resolution." In: *ITP'13*. Vol. 7998. LNCS. Springer, 2013, pp. 229–244. DOI: 10.1007/978-3-642-39634-2_18.

[WHJ14] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. "DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs." In: *Theory and Applications of Satisfiability Testing (SAT)*. Vol. 8561. LNCS. Springer, 2014, pp. 422–429. DOI: 10.1007/978-3-319-09284-3_31.

[Wir71] Niklaus Wirth. "Program Development by Stepwise Refinement." In: *Commun. ACM* 14.4 (1971), pp. 221–227. DOI: 10.1145/362575.362577.

[Wri95] Andrew K. Wright. "Simple Imperative Polymorphism." In: *Lisp and Symbolic Computation* 8.4 (1995), pp. 343–355.

[Xia+20] Li-yao Xia et al. “Interaction trees: representing recursive and impure programs in Coq.” In: *Proc. ACM Program. Lang.* 4.POPL’20 (2020), 51:1–51:32. DOI: 10.1145/3371119.

[Yan+11] Xuejun Yang et al. “Finding and understanding bugs in C compilers.” In: *PLDI’11*. ACM Press, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.

[ZYX14] Jie Zhang, Feng Yuan, and Qiang Xu. “DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans.” In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 153–166. ISBN: 9781450329576. DOI: 10.1145/2660267.2660289.

[Zha+13] Jie Zhang et al. “VeriTrust: Verification for Hardware Trust.” In: *Proceedings of the 50th Annual Design Automation Conference*. DAC ’13. Austin, Texas: Association for Computing Machinery, 2013. ISBN: 9781450320719. DOI: 10.1145/2463209.2488808.

[ZM03] Lintao Zhang and Sharad Malik. “Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications.” In: *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 2003, pp. 10880–10885. DOI: 10.1109/DATE.2003.10014.