Developing and certifying in Coq/MathComp of Datalog optimizations for network verification

Pierre-Léo Bégay

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Pierre-Léo BEGAY

Thèse dirigée par **Jean-François MONIN**, professeur Polytech Grenoble, Université Grenoble Alpes,
et codirigée par **Pierre CREGUT**, Ingénieur R&D Orange Labs

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Développement et certification en Coq/MathComp d'optimisations Datalog pour la vérification réseau

Development and certification in Coq/MathComp of Datalog optimizations for network verification

Thèse soutenue publiquement le 23 novembre 2021,

devant le jury composé de :

Mme Évelyne CONTEJEAN
Directrice de recherche, CNRS-Université Paris Sud, Rapporteur
Mr Damien POUS
Directeur de recherche, CNRS-ENS Lyon, Rapporteur
Mr Stéphane DEVISMES
Maître de conférences, Université Grenoble Alpes, Membre
Mme Stefania Gabriela DUMBRAVA
Maître de conférences, ENSIIE & Institut Polytechnique de Paris, Membre
Mr Stéphane GRUMBACH
Directeur de recherche, Inria Lyon, Membre
Mr David MONNIAUX
Directeur de recherche, CNRS-Université Grenoble Alpes,
Président du jury

Remerciements

Il me semble qu'il ne serait pas raisonnable de ne pas attaquer ces remerciements par ~~une double negation~~ celles et ceux qui ont pris sur leur temps pour venir m'écouter, voire lire le reste de ce document. Merci donc d'abord à Évelyne Contejean et Damien Pous pour leur minutieux travail de relecture. Du reste, les différents contacts que j'ai eus avec vous deux au cours de la thèse m'ont souvent aidé quand j'étais perdu dans mes doutes, merci pour ce soutien providentiel.

Merci également à Stéphane Devismes, Stefania Gabriela Dumbrava, Stéphane Grumbach et David Monniaux d'avoir accepté de faire partie de ce jury, votre présence m'honore.

Deux autres malheureux ont dû lire le présent manuscrit, et accessoirement encadrer le travail de thèse qu'il résume. Merci donc à Pierre Crégut et Jean-François Monin d'avoir accepté de travailler plusieurs années avec moi pendant tout ce temps, d'avoir mis en place un cadre de travail chaleureux mais sérieux (ou l'inverse), et surtout de m'avoir fait confiance du début à la fin. Je doute que vous réalisiez combien j'ai appris et mûri au cours de cette thèse.

Merci à Marie-Calixte de m'avoir toujours à la fois soutenu et aidé à me remettre en question, nos discussions ont (presque) systématiquement été d'une grande stimulation intellectuelle. Pour avoir *challengé* et étendu mon horizon culturel, je me dois également de remercier Stéphane et Prudence, discuter de cinéma ou jeu vidéo avec des badauds est généralement une torture quand on a la chance de vous connaître et vous fréquenter.

Kahina, sache que tu as été et restes un modèle pour moi. J'espère que tu me croiras si je te dis que je chante ta légende dès que j'en ai l'occasion, et que j'ai été très fier de faire une partie de mes études avec toi.

Ismail, toi tu n'es pas mon modèle, mais ton humour et ta bonne humeur (ainsi que ta capacité à toujours retomber sur tes pattes) forcent le respect – un authentique "élève fort sympathique". J'ai rarement la chance de vous voir Charlotte et toi, mais chaque occasion est un moment délicieux, merci donc pour chacun d'eux, et pour m'avoir fait découvrir le *laser game*. Dans la continuité, merci Mathias pour l'*esprit* et la joie de vivre que tu as apportés à l'homonerie. Je suis sincèrement impressionné par ton début de carrière académique et te souhaite le meilleur pour la suite.

Camille, j'ai appris avec émotion il y a quelques heures que tu étais devenue maman. C'est évidemment avec beaucoup de nostalgie que je te souhaite énormément de bonheur dans cette nouvelle étape de ta vie, en plus bien sûr de te remercier pour la découverte du foyer viet' et de l'arobase café.

Téo, Sophie, Alexandre, Juliette et Andres, merci d'avoir été de si bons mentors et amis (et un peu moins merci pour mon "traumatisme de tarot"). Kévin, tu fais de la crypto mais es un chic type. Merci de défendre cette discipline amusante mais quand même pas très sérieuse par rapport aux méthodes formelles.

Si je n'ai pas exactement étudié avec eux, ces années n'auraient pas été les mêmes sans Arnaud Durand et Paul Rozière. Ils m'ont fait confiance pour intégrer la licence maths-info de l'Université Paris 7, et soutenu face aux absurdités administratives qui ont failli m'en faire partir. Je leur dois l'infini bonheur de ces années.

En repensant aux années Paris 7, j'ai évidemment une pensée pour Jade. Nos jeux et ta complicité me manquent.

Jules, merci de m'avoir rappelé que MK2 Bibliothèque s'écrit avec un B majuscule.

Merci à Cécile d'avoir donné une dimension nouvelle à ma vie parisienne, tant sur les plans sociaux que culturels, et même un peu culinaires. J'hérite de toi une partie de celui que je suis aujourd'hui, et à mon avis pas la pire. Un sincère merci pour avoir veillé sur moi.

Alexis, j'aurais adoré faire ma thèse dans la même ville – allez, soyons fous, dans la même institution – que toi. Il aura finalement fallu se contenter d'un week-end à Strasbourg, pendant lequel j'ai retrouvé les passionnantes discussions qui nous animaient pendant nos études respectives. Ta curiosité et ton empathie forcent toujours le respect, et, ça ne gâche rien, ton appartement était vraiment super bien placé.

Hélène, j'aurais adoré faire un deuxième master dans la même ville – allez, soyons fous, dans (plus ou moins) la même institution – que toi. Je dois même confesser avoir été quelque peu jaloux en voyant la tournure prise par tes études et le début de ta vie professionnelle, mais la conception, le développement et la sortie d'un jeu vidéo ont finalement l'air d'être un chantier tout aussi pharaonique – à l'échelle humaine – qu'une thèse. J'espère que, maintenant que nous pouvons tous les deux revenir d'exil, nous pourrons reprendre les sorties ciné exotiques qui font notre légende.

Jessica, nos contacts sont globalement aussi incongrus et drôles que tu l'es toi-même, et j'ai hâte de venir *bruncher* avec toi à Grenoble. Clément, merci d'avoir toujours fait attention à mes cheveux, y compris quand je n'y avais pas touché.

Diane, ta rencontre est peut-être la plus aléatoire que j'aie jamais faite. Merci donc de m'avoir écouté – envers et contre tout, notamment le bon sens – parler de vérification de logiciel embarqué dans des avions, nous avoir fait découvrir Tag (non), m'avoir crédité comme conseiller scientifique de ton court-métrage (encore moins) et pour la magnifique surprise que tu m'as faite pour mes 26 ans. Bref, merci de mettre un peu d'animation dans ma vie.

Ju et Lucy, merci de m'avoir accueilli chez vous, et pour les découvertes culturelles que je n'aurais sans doute jamais faites ailleurs. Michiru, te croiser furtivement à un anniversaire fut un crève-coeur, j'espère avoir un jour l'occasion de reprendre un verre avec Diane et toi.

Mes trois années à l'ENS Cachan ont signé pour moi la fin de l'innocence – en tout cas des quatre sorties ciné hebdomadaires. Si j'y ai survécu, c'est avant tout grâce aux enseignants et camarades que j'y ai eus.

Un merci tout particulier à Itsaka et Théo qui, non contents de m'avoir supporté à l'époque, ont été mon *safe space matheux* pendant ces trois années de thèse. Ils m'ont écouté me plaindre, m'ont soutenu et partagé les absurdités administratives de leurs écoles doctorales ou institutions respectives (les anecdotes sur le MPI-SP s'annoncent d'ailleurs savoureuses). Merci plus spécifiquement à Théo et Laura de m'avoir fait découvrir Nantes, où j'ai fantastiquement bien mangé, et à Itsaka d'avoir été un si bon binôme en master, et notamment de m'avoir suivi à Jussieu.

Théis, des salles de sport américaines aux restaurants éthiopiens du $5^{ème}$, on aura sans doute plus voyagé et expérimenté qu'on aurait pu l'imaginer à l'époque des TDs de maths discrètes. Repenser à toute cette nourriture ingurgitée me rend mélancolique.

Olivier et Lucie, j'ai été sincèrement très honoré et touché d'être invité à votre mariage (et à l'enterrement de vie de garçon associé). Encore aujourd'hui, je garde maints souvenirs émus de la soirée, par ailleurs fort bien organisée, et vous renouvelle mes voeux de bonheur. Un grand merci pour votre amitié fidèle.

Victor, le plan capillaire mis à part, tu es la stabilité incarnée. Ton amour d'Ocaml et (surtout) ta présence à l'IRIF – ça inclut Niols en train de jouer à Borderlands tout en corrigeant un papier ou un dev Unix dans le coin opposé du bureau – ont été des constantes très *réconfortantes* pendant

ces trois années de thèse. Il va sans dire que je garde également en tête l'incroyable bar lyonnais dans lequel nous avons échoué (le mot est faible) en 2018.

Comment évoquer ces années sans mentionner les personnages que sont Charlie et Paul ? L'ambiance n'aurait pas été la même sans vous, votre incroyable bonne humeur et votre génie. En particulier, merci à Charlie d'avoir pris en charge la partie horrible du projet de programmation 2, et à Paul (et Athénaïs) pour sa délicieuse attention lors de mes 26 ans.

Athénaïs justement, merci à Gwendoline et toi pour la solidarité et l'ambiance à l'IRIF. Quelle belle époque.

Louise, je n'écris ou prononce jamais un mot en "coloc" sans penser à toi. Merci de m'avoir corrigé jusqu'à ce que ça imprime, et de m'avoir accueilli dans une situation critique. Je suis très heureux que tu aies été traînée au guet-apens au cours duquel nous nous sommes rencontrés.

Quentin 1, j'ai mis du temps à te cerner, mais *the juice was worth the squeeze*. Tu es pour moi un *safe space* à toi tout seul, et m'as permis de surmonter bien des soirées périlleuses. Merci pour tous ces moments.

Quentin 2, j'ai mis du temps à te cerner, mais *the juice was worth the squeeze*. Tu es pour moi un *safe space* à toi tout seul, et m'as permis de surmonter bien des soirées périlleuses. Merci pour tous ces moments.

Océane, merci de m'avoir toléré et partagé tes délicieux cookies.

Amina, merci de m'avoir permis de goûter à l'enseignement, je ne sais pas si j'aurais eu le courage de m'y lancer sans toi. Merci également pour ~~ton accueil~~ l'accueil de Denis à Lyon.

Léo & Léo, vous êtes absolument tordants, il me semble donc inconcevable que tous les groupes du MPRI n'aient cherché à vous récupérer, et je suis ravi que nous ayons gagné les enchères. Un merci tout particulier à Léo pour son accueil si chaleureux à Bruxelles, nous gardons un souvenir émerveillé de notre séjour.

En revanche, passer trois ans à Lannion fut une expérience quelque peu traumatisante, mais j'y ai heureusement fait la rencontre de personnes fantastiques qui m'ont permis d'y survivre. Merci donc à Alice, Sergio et David. Partager ma cuisine, mon bureau ou un couscous avec vous fut systématiquement source de joie, et c'est peu dire que votre compagnie quotidienne me manque aujourd'hui. Merci également à Patrick et Dom pour leur accueil, ainsi qu'à La Medina et Le Gringo, clairement les deux endroits les plus cool de la ville.

J'ai bien sûr rencontré moult enseignants au cours de mes études supérieures, dont beaucoup sont de fantastiques pédagogues, voire des modèles dans ma propre pratique de l'enseignement. Sans être exhaustif, j'ai une pensée pour David Baelde, qui m'a soutenu jusqu'au bout et à qui j'ai volé sa super posture au tableau, pour Claudine Picaronny, véritable modèle de clarté aux devoirs toujours passionnants, pour Hubert Comon-Lundh, qui nous a enseigné la rigueur, ou Carlos Geraci, dont le cours de linguistique de langue des signes fut d'un grand vertige humain et intellectuel.

J'ai également une pensée pour Rached Mneimné et Gilles Dowek, qui m'ont appris qu'un cours de maths peut être à la fois terriblement précis et merveilleusement humain, et pour Timo Jolivet et Irène Marcovici, qui m'ont poussé à aller plus loin, et sans lesquels je ne saurais sans doute pas aujourd'hui ce dont je suis capable.

Enfin, je remercie Jean Goubault-Larrecq pour ses innombrables et incroyables anecdotes, qui me permettent aujourd'hui de briller dans les dîners en ville.

L'expérience fut plus courte, mais je tiens aussi à remercier l'équipe de l'EJCP pour son accueil d'une très grande qualité. Cette semaine à Strasbourg m'aura fait le plus grand bien.

J'ai eu la chance de pouvoir moi-même enseigner pendant la thèse. La tâche est (étrangement) bien résumée par Renaud Van Ruymbeke dans ses mémoires : "Instruire, c'est d'abord écouter et dialoguer ; ce n'est pas juger, mais chercher, formuler des hypothèses tout en se méfiant de ses intuitions, remettre en question, comprendre et, enfin, démontrer et expliquer. Le dialogue est essentiel".

Merci à Adeline, Alice (x2), Ahmed, Beyza, Camille, Chinatsu, Chloé, Cindy, Edouardo, Elise, Flora, Inae, Jiaqi, José, Maï-Ly, Mana, Margot, Mathilde, Navaoreethi, Nicolas, Ninoh, Orlando, Salomé, So-Hoon, Tiffany, Yanis et bien d'autres pour ce dialogue, et plus généralement d'avoir été de fantastiques étudiants et étudiantes. Vous voir apprendre (ou non), vous amuser (ou non), exercer votre curiosité (ou non) et progresser (ou non) fut non seulement intellectuellement très stimulant, mais surtout une grande source de fierté et de bonheur pour moi. Il va sans dire que je vous souhaite tout le meilleur pour la suite de vos études et vies respectives.

La thèse n'est pas le début de l'aventure (on fait en général un master avant), mais elle n'en est pas non plus la fin. Merci donc pour leur accueil à Stéphane, Olivier, Tony, Céline, Narjes, Etienne, ~~Michel~~ Eric, Erika, et tous les autres que j'ai désormais la chance d'avoir comme collègues.

Alex, Elina et Maël, merci pour votre fidélité, votre écoute, et tous ces week-ends normands. Le Havre n'est peut-être pas la plus belle ville de France (ni la deuxième ou la troisième), mais y aller est systématiquement pour nous une sacrée expérience.

Jules est une personne tellement extraordinaire qu'il serait insensé de ne pas le faire apparaître deux fois dans ces remerciements. Cependant, toute personne me connaissant sait qu'il serait virtuellement impossible d'énumérer tout ce que je lui dois, et que l'amitié que je lui porte est de toute façon depuis longtemps inconditionnelle.

Un grand merci à Sylvie, Hervé, Lucien et Gabin, bien sûr pour m'avoir accueilli dans leur famille comme si j'en avais toujours fait partie, mais aussi et surtout de m'avoir fait découvrir les fouées.

Impossible de ne pas rendre hommage à mes parents, qui n'ont eu de cesse d'oeuvrer à mon bonheur. Si ce dernier ne se résume – heureusement – pas à la bonne conduite d'une thèse, force est de constater que l'éducation qu'ils m'ont donnée fut déterminante dans la préservation de ma santé mentale pendant ces années.

Merci à Julie d'être venue voir Suspiria ce mercredi de juin 2018, et un très, très grand merci à tous les autres de s'être absentés.

Enfin, un sincère merci à Hafça et Zeinab, sans qui ma vie serait aujourd'hui bien différente – et sans doute substantiellement moins rigolote.

General-audience abstract

A 1995 Wired article titled "How Anarchy Works" states that "Part of what has made the Net successful is precisely that: it works". This is a good reflection of the general philosophy behind network engineering, which is much more built *via* concrete experimentations than formal reasoning.

Although efficient, this approach did not scale with the exponential digitalization of the last decades. Network failures, which are bound to happen with the level of care historically put into network design, are becoming more and more costly, if not critical.

This situation led to the introduction of complex, but much sounder methods – generally refered to as *formal methods* – to networking ten to fifteen years ago. However, even the tools brought by this line of thought sometimes contain gaping holes that need to be addressed. This thesis identifies a tool that reaches a reasonable efficiency thanks to dubious shenanigans, and presents a formally defined and verified alternative.

Résumé pour le grand public

Un article publié par Wired en 1995 intitulé "Comment marche l'anarchie" explique que "l'une des raisons du succès de l'internet est simplement le fait que ça marche". Cette phrase reflète bien la philosophie générale derrière internet, qui s'est bien plus construit *via* l'expérimentation que le raisonnement formel.

Bien qu'efficace, cette approche n'a pas suivi l'exponentielle numérisation des dernières décennies. Les pannes réseaux, qui ne peuvent qu'arriver avec le niveau de soin historiquement mis dans la conception de réseaux, sont de plus en plus coûteuses et critiques.

Cette situation a mené à l'introduction de méthodes – généralement appelées *méthodes formelles* – plus complexes, mais également plus sûres dans le monde du réseau. Cependant, même les outils issus de cette philosophie peuvent contenir des angles morts. Cette thèse identifie un outil qui atteint une efficacité raisonnable grâce à de douteux procédés, et présente une alternative formellement définie et vérifiée.

Cette thèse est écrite en anglais, mais l'annexe C en propose un résumé en français.

Contents

Part I
Introduction

Donc si vous décidez d'écrire cet article, je vous colle au violon pour incitation à l'émeute, mensonge, trouble de l'ordre public, folie paranoïaque, tentative de suicide et prose pitoyable

---

Lewis Trondheim, *Les formidables aventures de Lapinot* (tome 3, *Walter*)

Context and motivation

Over the last decades, the world has gone more and more digital. This trend was not refuted in 2020 or 2021, as professional and personal services are increasingly provided and accessed through computers, tablets and mobile devices. This intense *digital shift* means that network failures are more costly and prejudicial than ever[1], let alone critical in many instances[2]. We emphasize that the failures we mention and are interested in do not result from external attacks – which do occur on a weekly, if not daily basis and in industrial proportions –, but are to be seen as bugs.

These bugs stem, first and foremost, from the extremely high complexity of network design, which in turn comes from the intrinsically distributed nature of networks. Moreover, networking has run for a long time on a *duct tape culture*, in the sense that it lacked formal foundations, and the possibilities that the existence and study of such foundations unlock.

Over the last ten to fifteen years, researchers with a background in programming language theory have started to take an interest in networking, and how they could apply their theoretical tools and approaches to this field. Combined with the critically increased need for safety (and security), this situation led to the introduction of formal methods for networks. This trend is also fostered by the latest advances in formal methods, both in terms of modeling techniques and computational efficiency (e.g. the existence of fast solvers such as Z3).

One such tool that was introduced is Network Optimized Datalog (NoD), a Datalog engine developed at Microsoft and tailored to handle programs that describe, in the form of Horn clauses, the behavior of a particular network. Although an interesting step in the desired direction, using this engine requires engineers to manually write encodings of each analyzed network, which in itself is a complex and error-prone process.

Moreover, NoD does not scale with *naive* translations of real-size networks. In practice, the authors work with programs that contain many inlined values, using manual, convoluted, undocumented and unjustified Datalog-level program transformations. This gap in an otherwise remarkable tool led us to work on the design and automatization of such program transformations, this time with a full formalization.

However, having a formalization of non-trivial operations is not enough to trust them. The aim of our work has then been the formal verification of these transformations in the Coq proof assistant, using (and slightly extending) an existing Coq implementation of Datalog.

Although inspired by network verification, our work is not circumscribed to it. Concretely, the analyses and rewritings we provide can be used – and relevant – in other contexts. Moreover, we believe that this works brings a new insight into the semantics and formal study of Datalog programs, which may serve as the basis of future works in other contexts.

[1] https://techcrunch.com/2020/07/17/cloudflare-dns-goes-down-taking-a-large-piece-of-the-internet-with-it/
[2] https://www.reuters.com/business/media-telecom/orange-blames-network-outage-software-failure-audit-2021-06-11/

Contribution(s)

Questions and results

The starting point of this thesis was the identification and analysis of a caveat in the Network Optimized Datalog engine in the presence of primitive predicates with multiple variables. To address this issue, we designed a static analysis for Datalog, as well as two program transformations that leverage it. Both the static analysis and the transformations have been verified in the Coq proof assistant, using a previously introduced Coq formalization and implementation of Datalog.

Our work required and led to the extension of some tools, mainly the introduction of a trace semantics for Datalog and its verified implementation in the aforementioned Coq formalization of Datalog. We also develop some new finite types for the Mathematical Components library, upon which this formalization relies.

Finally, we present a tighter version of our static analysis and show that it is not fit for every Datalog program. This leads us to try to caracterize the precise class of Datalog programs that supports it, but our intuition is yet to be formally verified.

List of publications

The work on this thesis resulted in the following contributions to the scientific discussion:

- a talk at the 2020 Coq workshop on the development of new finite types for the Math-Comp library [Bégay et al., 2020a]
- a paper at the $19^{èmes}$ *journées approches formelles dans l'assistance au développement de logiciels* (AFADL 2020) conference [Bégay et al., 2020b]
- a paper at *Certified Programs and Proofs* (CPP) 2021 [Bégay et al., 2021]

Toolkit

The work presented in this dissertation has been performed using a variety of existing tools. The details of some (e.g. Datalog, Network Optimized Datalog, DatalogCert) play an important part in the rest of this document, meaning that they are fully introduced in subsequent chapters.

On the other hand, some tools are either too rich and well-known (i.e. Coq or Z3) or secondary (i.e. OpenStack) to require such a detailed presentation, and will be considered strictly from a user-perspective. In particular, we will not delve into their theoretical foundations or implementations, although some references are provided. These tools and the part they played in our work are outlined below.

Building theorems and proofs – Coq & MathComp

Coq is a well-established proof assistant, that contains a functional programming language as well as logical tools to reason about the developed programs and, more generally speaking, formaly defined systems. It has been used with much success in the development and verification of compilers [Appel and Blazy, 2007, Chlipala, 2010, Kumar et al., 2014, Letan and Régis-Gianas, 2020, Bodin et al., 2018], most notably the verified C compiler CompCert [Leroy, 2009, Appel et al., 2014]. It is also the framework in which we develop and verify the contributions presented in this document.

Another area where Coq has shined is the formal proof of more traditional mathematical results [Cruz-Filipe et al., 2004, O Connor, 2005, Bauer et al., 2017, Zsidó, 2013, Makarov and Spitters, 2013, Beeson et al., 2018, de Rauglaudre, 2017]. One of the main achievements in this field is the formal and verified proof of the four colour theorem [Gonthier, 2007], which led to the development of a new library, called Mathematical Components, or MathComp [Gonthier et al., 2016], which we used extensively in the course of our work.

This dissertation assumes some familiarity with Coq, or at least another proof assistant, but not MathComp. In the eventuality that it should find its way into the hands (or screen) of someone with no prior knowledge of a proof assistant, Appendix A provides an introduction to Coq, from a very practical and user's perspective. This light introduction may not be exhaustive enough to follow the full details of our work, but should be sufficient to get the grasp of the main ideas behind it. A much more complete presentation of Coq, including theoretical foundations, can be found in [Castéran and Bertot, 2004]. The subset of MathComp relevant to this work will be introduced in Section 3.1.

Coq is not the only proof assistant or theorem prover available, as it coesxists with the HOL family (Isabelle/HOL [Nipkow et al., 2002], HOL-Light [Harrison, 2013]),

PVS [Owre et al., 1992], Agda [Norell, 2008], Nuprl [Constable et al., 1986] or Matita [Asperti et al., 2011]. It is however a mature tool, with strong – and familiar – theoretical foundations (the Curry-Howard correspondence), a vast community and a remarkable track record of formalizations and certifications. Moreover, we saw the recent introduction of a Coq formalization and implementation of Datalog [Dumbrava, 2016] as an opportunity to build our work upon solid bases, and possibly make it part of an ongoing, larger project[3]. This formalization is formally introduced in Chapter 3.

Solving first-order riddles – Z3

Z3 [de Moura and Bjørner, 2008] is a Satisfiability Modulo Theory (SMT, [Monniaux, 2016]) solver developed at Microsoft. Basically, it expects a theory T expressed as a set of first-order logic formulae as well as a formula ϕ, and tries to check the satisfiability of ϕ w.r.t. T. Z3 relies on many heuristics, sometimes domain-specific [Hoder et al., 2011, Hoder and Bjørner, 2012, Bjørner et al., 2015, Wintersteiger et al., 2013], which make it one of the most efficient SMT solvers available [Weber et al., 2019].

First-order logic is one of the building blocks of theoretical computer science, it is then natural that a practical tool which handles it efficiently would serve as a backend for many other projects. Such examples include the Boogie intermediate language for imperative programs [Barnett et al., 2005] (upon which other tools are in turn built), the verification-oriented functional programming language $F^\star$ [Swamy et al., 2013], or the symbolic execution engine Klee [Cadar et al., 2008]. It is also the backend of the Network Optimized Datalog engine [Lopes et al., 2015], which is the topic of Chapter 5.

Managing a network – OpenStack & Neutron

OpenStack [Sefraoui et al., 2012] is a cloud platform, i.e. a set of tools that abstract the resources used for some service (e.g. computing, storage or network). A classical, general-audience example of cloud platform is Dropbox, which provides storage to users who do not have to worry about the actual handling of their files (e.g. which physical drive they are on, how they are spread and so on), and are supplied a simple, minimalistic interface.

This notion of abstraction is also found in networking, where a physical network, also called *underlay*, is a physical infractructure upon which multiple *overlay* networks can be layered simultaneously. For example, a virtual private network (VPN) is built upon another, existing network, using tunneling protocols to work in isolation from the latter.

Concretely, OpenStack can be used to configure, deploy and maintain such virtual networks. It is a popular, open-source and extensible tool, meaning that, in practice, it comes with a very high combinatorics of configurations – one can think of them as a matrix of features and implementations.

On one hand, the abstraction layer introduced by OpenStack opens the way to automatic verification, but on the other hand, the modelization must be able to handle the many supported configurations. The Network Oriented Datalog verification tool, which computes the properties of a specific network using its low-level description, rather than the specification of higher-level network properties such as reachability, lacks this genericity.

[3] http://datacert.lri.fr/

This led Pierre Crégut, R&D engineer at Orange Labs and co-advisor of this work with Jean-François Monin, to work on a higher-level network verification tool built upon Network Optimized Datalog, called Octant. Unlike NoD, Octant separates the specification of general network properties and the description of specific networks, in the sense that it expects the former and checks it against the latter.

Octant uses a compotent of OpenStack called Neutron, which defines its mission as "provid[ing] on-demand, scalable, and technology-agnostic network abstraction"[4]. More concretely, Neutron can be used to easily extract low-level informations about the network, such as the forwarding tables of switches, in an structured manner. Octant uses it to fetch the implementation of the analyzed network, and provides these informations to the deduction engine.

The work described in this dissertation has been integrated to Octant, which is described more in-depth in Chapter 6.

[4] https://wiki.openstack.org/wiki/Neutron

Outline

This dissertation is split into four parts, excluding the introduction (Part I) and conclusion (Part VI). The work presented being at the intersection between two different research areas (network verification and programming language theory), the first two parts are dedicated to the introduction of relevant tools and concepts from both. The other two parts then introduce and discuss our contributions.

Part II addressses the networking component of this work. First, it provides some context by outlining existing approaches in network verification. It then focuses on the Network Optimized Datalog (NoD) and Octant tools, which were the starting points of our work. In particular, we explain how the latter builds upon the former to provide more genericy, but is limited in doing so by the internals of NoD.

Part III first recalls the building blocks of first-order logic, and then uses them to formally define the Datalog logic programming language. Finally, it outlines a previously existing verified implementation of Datalog within the Coq proof assistant. This in-depth formalization of Datalog will serve as the reference specification and implementation for the rest of our work.

Part IV discusses why we had to extend some tools, and how it was done. More concretely, we introduce a trace semantics for Datalog and certify it in Coq, and develop some new finite types for the MathComp library.

Finally, Part V presents the static analysis and the two rewritings we designed, as well as their verification. It also contains a general discussion on these points, including the lessons learned from the Coq certification process.

Part II

Verified implementation of a logic programming language: Datalog

This thesis being about the logic programming language Datalog, we need to formally define it. To do so, we first recall the inner workings of first-order logic, upon which Datalog is built, in Chapter 1. We then move on to the pen and paper definition of Datalog in Chapter 2. Since our work is developed on top of a Coq formalization of Datalog [Dumbrava, 2016], we reuse their definitions – which are themselves based upon [Lloyd, 1987b] –, although some minor modifications are made. We finally present in Chapter 3 the core of [Dumbrava, 2016], i.e. their Coq formalization of Datalog [Benzaken et al., 2017b].

Chapter 1

First-order logic

> La vie serait tout de même beaucoup plus simple si tout le monde s'exprimait en logique du premier ordre
>
> Zeinab Galal, conversation privée

In his *leçon inaugurale* at the *Collège de France* [Leroy, 2019], Xavier Leroy emphasizes on the central and paramount role of logic throughout computer science[1]. Most logical tools are built upon first-order logic, which is also fundamental in database theory.

The core feature of databases is to combine informations it stores to answer queries provided by a user. In traditional database theory, the dialog between user and database, in particular the formulation of query, relies on the logic-based relational calculus [Codd, 2002] – although it should be noted that some modern approaches to databases, such as NoSQL, shift away from logic to focus on graph-based or object-oriented methods [Dean and Ghemawat, 2008, Abiteboul et al., 2011] –, as well as unification mechanisms that can be traced back to the early works of Jacques Herbrand [Herbrand, 1930].

The combination of informations is akin to logical deduction, and in particular relevance of the resolution principle [Robinson, 1974] was early noted [Minker, 1988, Kuhns, 1967, Levien and Maron, 1965, Green and Raphael, 1967]. Going further, [Van Emden and Kowalski, 1976] introduced the foundations of logic programming, which mechanized even further the use of logic in database settings. This line of work eventually produced the Datalog programming language, which will be studied in this document after the basics of first-order logic are recalled.

Sections 1.1 and 1.2 formalize its syntax and semantics, respectively. Then, Section 1.3 introduces the technical but useful concept of normal form, and Section 1.4 presents some inference systems within first-order logic.

1.1 Syntax

This Section formalizes the syntax of first-order logic. To do so, we first go over the way first-order logic formulae are built, and then how they are manipulated and transformed to be reasoned about.

[1] "La logique ! On y revient encore et toujours ! C'est le *leitmotiv* de cette leçon"

1.1.1 Building blocks

First-order logic (FOL) usually brings to mind its well-known quantifiers, but a thorough study of the topic starts with the atomic, non-logical elements of the language. These are defined in the framework of so-called signatures.

Definition 1.1. A first-order **Signature** is a triple $(\mathcal{F}, \mathcal{P}, ar)$. $\mathcal{F}$ is a set of function symbols, that are used to build the arguments for the predicates. The predicate symbols are found in $\mathcal{P}$, and these two sets are disjoint. The good use of these function and predicate symbols is insured by the $ar : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ function, which assigns an arity to every symbol.

Notation 1.2. For clarity, the function and predicate symbols can be augmented with their arity. For example, given the f and p symbols with $ar(f) = n$ and $ar(p) = m$, we can replace f and p by $f_{/n}$ and $p_{/m}$. In that setting, the signature is simply defined as $\Sigma \equiv (\mathcal{F}, \mathcal{P})$.

Notation 1.3. A function symbol f such that $ar(f) = 0$ is called a **constant**, and the set of constants is written $\mathcal{C}$. A predicate symbol p such that $ar(p) = 0$ is called a propositional variable.

Example 1.4. Peano arithmetic is defined upon signature $(\mathcal{F} = \{0_{/0}, s_{/1}, +_{/2}, \times_{/2}\}, \mathcal{P} = \{=_{/2}\})$. In that setting, an expression such as $(1 + 2) \times 3$ is translated as $\times(+(s(0), s(s(0))), s(s(s(0))))$, which does enforce the arity constraints.

On top of the vocabulary – i.e. predicate and function symbols, as well as constants – of our first-order language $\mathcal{L}$, we can introduce the grammar of first-order logic *via* the additional symbols:

- a countable set of variables $\mathcal{X}$;
- the existential and universal quantifiers, respectively denoted as $\exists$ and $\forall$;
- the connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence), as well as the (nullary) truth symbols $\top$ (true) and $\bot$ (false);
- parentheses and punctuation.

Now that we have all the relevant symbols, we can build up to the actual first-order formulae, starting with the words of the language, called terms:

Definition 1.5. The set of **$\mathcal{L}$-terms** is the minimal set $T_\Sigma(\mathcal{X})$ that contains the variable set $\mathcal{X}$, and satisfies

$$\text{for any } f_{/n} \text{ in } \mathcal{F}, \text{ if } t_1, ..., t_n \text{ are in } T_\Sigma(\mathcal{X}), \text{ then so is } f(t_1, ..., t_n)$$

In other words, the terms are the constants, the variables, and the function symbols applied using a number of terms corresponding to their arity.

Example 1.6. Reusing the signature of Example 1.4 with the variables $\mathcal{X} = \{x, y\}$,

- x, a variable, is a term;
- 0, a constant, is a term;

- $s(0), \times(y, s(0))$, and $+(x, \times(y, s(0)))$, *via* three successive applications of the recursive rule, are terms.

The terms are then used in conjunction with predicate symbols to build so-called atoms, which are then enriched to form the set of base sentences of the language, called atomic formulae.

Definition 1.7. An **atom** is a predicate symbol applied to a number of terms corresponding to its arity, i.e. $p(t_1, ..., t_{ar(p)})$ with $p \in \mathcal{P}$ and $t_i \in T_\Sigma(\mathcal{X})$ for all i.

Definition 1.8. An **atomic formula** is either an atom, or one of the two special symbols $\top$ and $\bot$, also called *true* and *false*, respectively.

We now have all the building blocks that can be combined with the logical symbols to build the actual first-order formulae.

Definition 1.9. The set of $\mathcal{L}$-formulae is the minimal set $F_\Sigma(\mathcal{X})$ that contains the atoms built upon $\mathcal{P}$ and $T_\Sigma(\mathcal{X})$ and satifies the two following rules:

- the $\mathcal{L}$-formulae are stable under binary logical connectors, i.e. if ϕ_1 and $\phi_2 \in F_\Sigma(\mathcal{X})$, then $\phi_1 \square \phi_2 \in F_\Sigma(\mathcal{X})$, with $\square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.
- the $\mathcal{L}$-formulae are stable under negation and quantification, i.e. if $\phi \in F_\Sigma(\mathcal{X})$, then $\neg\phi$, $\forall x, \phi$ and $\exists x, \phi \in F_\Sigma(\mathcal{X})$, where $x \in \mathcal{X}$

Example 1.10. Using $\mathcal{X} = \{x, y\}$, $\mathcal{P} = \{=_{/2}\}$ and $\mathcal{F} = \{0_{/0}, s_{/1}, +_{/2}, \times_{/2}\}$, the following two sentences are first-order logic formulae:

- $\forall x, \exists y, = (+(x, \times(y, s(0))), 0)$

  Using infix and standard integer notations: $\forall x, \exists y, x + (y \times 1) = 0$

- $\forall x, \forall y, \Rightarrow (= (x, \times(s(s(0)), y)), = (s(s(x)), \times(s(s(0)), s(y))))$

  Using infix and standard integer notations: $\forall x, \forall y, x = 2 \times y \Rightarrow x + 2 = 2 \times (y + 1)$

The previous definitions are summed up in the following grammar:

$$\begin{array}{rcl} \text{Terms } t & ::= & x \in \mathcal{X} \mid c \in \mathcal{C} \mid f(t_1, ..., t_n), f \in \mathcal{F} \ \& \ ar(f) = n \\ \text{Atomic Formulae } A & ::= & \bot \mid \top \mid p(t_1, ..., t_n), p \in \mathcal{P} \ \& \ ar(p) = n \\ \text{Complex Formulae } \phi & ::= & A \mid \phi_1 \square \phi_2, \ \square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \mid \neg\phi \mid \forall x, \phi \mid \exists x, \phi \end{array}$$

This thesis focuses on the logic programming language Datalog, which only relies on a subset of first-order logic. One of the main restrictions is the exclusive use of clausal formulae.

Definition 1.11. A **literal** is a positive or negated atomic formula.

Definition 1.12. A **clause** is a disjunction of literals.

Remark 1.13. A clause $L_1 \vee ... \vee L_n$ can be written in an implicative but semantically equivalent form $(\neg L_1 \wedge ... \wedge \neg L_{i-1} \wedge \neg L_{i+1} \wedge ... \wedge \neg L_n) \rightarrow L_i$

One of the main restrictions of Datalog is its exclusive use of such clauses. Another restriction is the absence of function symbols, i.e. $\mathcal{F} = \emptyset$. The syntax of Datalog will be formally introduced in Section 2.1.

1.1.2 Manipulating formulae

The variables appearing in a first-order logic formula are a stand-in for many potential values. Reasoning about such formulae will then require tools to perform the replacement, or substitution, of variables by terms.

We build up to that concept, starting with a categorization of variables based on their quantification.

Definition 1.14. The set $FV(t)$ of **free variables** of a $\mathcal{L}$-term t is defined as:

- $FV(x) = \{x\}$, where x is a variable;
- $FV(f(t_1, ..., t_n)) = \bigcup_{i=1}^{n} FV(t_i)$, where all t_i are terms and f a function symbol.

Definition 1.15. A $\mathcal{L}$-term t is **ground**, or closed, if $FV(t) = \emptyset$, i.e. if the term contains no variable. The set of ground terms is written T_Σ.

Definition 1.16. The set of **free variables** of a first-order logic formula ϕ, written $FV(\phi)$ is defined with the following rules:

- The free variables of an atom are the variables appearing in it
  - $FV(\bot) = FV(\top) = \emptyset$
  - $FV(p(t_1, ..., t_n) = \bigcup_{i=1}^{n} FV(t_i)$, where all t_i are terms and p is a predicate symbol
- The free variables are propagated by the logical connectives
  - $FV(\phi \square \psi) = FV(\phi) \cup FV(\psi)$, where $\square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
  - $FV(\neg\phi) = FV(\phi)$
- The free variables are *canceled* by the quantifications
  - $FV(\forall x, \phi = FV(\phi) \backslash \{x\}$
  - $FV(\exists x, \phi = FV(\phi) \backslash \{x\}$

In other words, the free variables of a formula ϕ are the variables that do not appear directly *under* (in a syntactic sense) a quantification.

Definition 1.17. The **bound variables** of a $\mathcal{L}$-formula ϕ, written $BV(\phi)$, are the variables which appear directly *under* a quantification. They are computed using the following rules:

- There are no bound variables in an atomic formula, as there is no quantification either
  - $BV(\bot) = BV(\top) = BV(p(t_1, ..., t_n) = \emptyset$
- Just like free variables, the bound ones are propagated by the logical connectives

  - $BV(\phi \square \psi) = BV(\phi) \cup BV(\psi)$, where $\square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$;
  - $BV(\neg\phi) = BV(\phi)$

- The bound variables are introduced by the quantifications:
  - $BV(\forall x, \phi = BV(\phi) \cup \{x\}$
  - $BV(\exists x, \phi = BV(\phi) \cup \{x\}$

Definition 1.18. The set of **variables of a formula** ϕ, written $VAR(\phi)$, is the set union of $FV(\phi)$ and $BV(\phi)$.

Definition 1.19. An $\mathcal{L}$-formula is called **ground** if $VAR(\phi) = \emptyset$, i.e. if it contains no variable.

Definition 1.20. An $\mathcal{L}$-formula is called **closed**, or **a sentence** if $FV(\phi) = \emptyset$, i.e. if any variable appearing in the formula is bound (to a quantification). Intuitively, it means that there is no *loose* variable, i.e. a variable whose instantiation is not dictated by a quantifier, so that the meaning of the formula does not depend on the meaning of unbounded variables. In practice, terms and formulae will be interpreted by assining a value to each such variable. The set of $\mathcal{L}$-sentences is written $SEN_{\mathcal{L}}$.

Now that we can reason about the different types of variables in a formula, we can move on to the actual substitutions and their application.

Definition 1.21. A **(partial) substitution** σ is a mapping from the set of variables $\mathcal{X}$ to the terms $T_\Sigma(\mathcal{X})$. A substitution is represented as a list of the individual variable / term mappings, i.e. $[x_1 \mapsto t_1, ..., x_n \mapsto t_n]$.

Definition 1.22. Given a variable x and a substitution $\sigma = [x_1 \mapsto t_1, ..., x_n \mapsto t_n]$, the **instantiation** of x with σ, written $\sigma(x)$, or σx, is defined as

$$\sigma(x) = \begin{cases} t_i & \text{if } x = x_i \text{ for some } i \in [1, n] \\ x & \text{otherwise} \end{cases}$$

In other words, if x appears in σ, then the associated term is returned. Otherwise, the result is the variable itself.

Definition 1.23. The **domain**, or support, of a substitution $\sigma = [x_1 \mapsto t_1, ..., x_n \mapsto t_n]$ is the set $\{x \in \mathcal{X} \mid \sigma(x) \neq x\}$, i.e. the set of variables that appear (and are associated to a different term) in σ.

Remark 1.24. A substitution σ can also be extended to operate on terms, in which case the constants are left untouched and the complex terms are treated inductively. Such a $\overline{\sigma} : T_\Sigma(\mathcal{X}) \to T_\Sigma(\mathcal{X})$ can then be defined as

$$\begin{cases} \overline{\sigma}(c) = c & \text{with } c \in \mathcal{C} \\ \overline{\sigma}(v) = \sigma(v) & \text{with } v \in \mathcal{X} \\ \overline{\sigma}(f(t_1, ..., t_n)) = f(\overline{\sigma}(t_1), ..., \overline{\sigma}(t_n)) & \text{with } f \in \mathcal{F} \text{ and } t_i \in T_\Sigma(\mathcal{X}) \text{ for any } i \in [1, n] \end{cases}$$

We can now lift the notion of instantiation to first-order logic formulae.

Definition 1.25. The **instantiation of a first-order logic formula** using a substitution σ is defined inductively as

$$\begin{cases} \sigma(\bot) = \bot & \\ \sigma(\top) = \top & \\ \sigma(t) = \overline{\sigma}(t) & \text{where } t \in T_\Sigma(\mathcal{X}) \\ \sigma(\phi_1 \square \phi_2) = \sigma(\phi_1) \square \sigma(\phi_2) & \text{where } \square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \\ \sigma(\neg\phi) = \neg\sigma(\phi) & \\ \sigma(\square x, \phi) = \square x, \sigma \backslash [x \mapsto \sigma(x)](\phi) & \text{where } \square \in \{\forall, \exists\} \end{cases}$$

In other words, the substitutions work inductively on the formulae. Moreover, a quantification over a variable x removes any mapping of x that may have been present in the used substitution.

Finally, we introduce a notion of order over substitions.

Definition 1.26. Let us consider a signature without function symbols of non-zero arity, i.e. $\forall f \in \mathcal{F}, ar(f) > 0$. In that setting, the terms are restricted to variables and constants. We can now define a partial order on substitutions as:

$$\sigma_1 \preceq \sigma_2 \equiv \forall x, \sigma_1(x) \in \mathcal{C} \Rightarrow \sigma_1(x) = \sigma_2(x)$$

In other words, σ_2 is compatible with, and more precise than σ_1.

1.2 Semantics

As previosuly stated, a first-order language $\mathcal{L}$ is built over a signature $\Sigma = \{\mathcal{F}, \mathcal{P}, ar\}$, where $\mathcal{F}$ is a set of function symbols used to build terms, $\mathcal{P}$ is a set of predicate symbols used to build atoms, and ar is an arity function for both kinds of symbols. We first assume such a signature.

We need to specify the elements we are talking and reasoning about. This set is called the **domain of discourse**, or **universe**, and written $U_\mathfrak{M}$. Once we have such a universe, we can interpret in it the (syntactical) symbols of Σ using a $\Sigma - structure$.

Definition 1.27. A **Σ-Structure** $\mathfrak{M} = (U_\mathfrak{M}, I)$ consists of a non-empty universe $U_\mathfrak{M}$ and an interpretation function $I : \Sigma \to U_\mathfrak{M} \cup \{\top, \bot\}$, such that

- for every $f \in \mathcal{F}, I(f) : U_\mathfrak{M}^{ar(f)} \to U_\mathfrak{M}$
- for every $p \in \mathcal{P}, I(p) : U_\mathfrak{M}^{ar(p)} \to \{\top, \bot\}$

A Σ-structure is sometimes called Σ-interpretation or Σ-algebra.

In Chapter 2, the semantics of Datalog will be defined using a specific type of universe and Σ-structures, called Herbrand, where the syntactic part of the language is directly used for its interpretation.

Definition 1.28. Given a signature Σ, the **Herbrand Universe** $U_{\mathcal{H}}$ is the set of ground terms of the language, i.e. T_Σ.

Definition 1.29. A **Herbrand Σ-structure** $\mathcal{H} = (U_{\mathcal{H}}, I_{\mathcal{H}})$ is a Σ-structure based on a Herbrand universe.

A Σ-structure $\mathfrak{M}$ is used to evaluate the veracity of a first-order logic formula, i.e. assign a boolean value. To do so, we need to be able to assign values to variables.

Definition 1.30. A **valuation** ι over $\mathfrak{M}$ is a partial function $\mathcal{X} \to U_{\mathfrak{M}}$.

Remark 1.31. In Section 1.1.2, we defined the similar notion of substitution. However, unlike valuations, substitutions work on a strictly syntactic level, and are total functions – they associate a value to *every* variable.

Definition 1.32. The **extension of a valuation** ι, written $\iota[x \mapsto u]$ where x is a variable and u an element of the universe, works like ι with a special case when applied to x. More formally, given a variable y,

$$\iota[x \mapsto u]y = \begin{cases} u & \text{if } x = y \\ \iota(y) & \text{otherwise} \end{cases}$$

The interpretation now works homophorphically (or recursively) to assign an element of the universe to each term, and then a binary valuation to formulae.

Definition 1.33. The **interpretation of $\mathcal{L}$-terms** in $\mathfrak{M}$ under a valuation $\iota : \mathcal{X} \to U_{\mathfrak{M}}$ is defined as a mapping $[\![\cdot]\!]^{I,\iota} : T_\Sigma(\mathcal{X}) \to U_{\mathfrak{M}}$:

- $[\![x]\!]^{I,\iota} = \iota(x)$
- $[\![f(t_1, ..., t_n)]\!]^{I,\iota} = I(f)([\![t_1]\!]^{I,\iota}, ..., [\![t_n]\!]^{I,\iota})$

Definition 1.34. The **evaluation of $\mathcal{L}$-formulae** in $\mathfrak{M}$ under a valuation $\iota : \mathcal{X} \to U_{\mathfrak{M}}$ is defined as the mapping $[\![\cdot]\!]^{I,\iota} : SEN_{\mathcal{L}} \to \{0, 1\}$, using the following rules.

- $\bot$ and $\top$ correspond to 0 and 1, respectively:
  - $[\![\bot]\!]^{I,\iota} = 0$
  - $[\![\top]\!]^{I,\iota} = 1$
- The evaluation of an atom is done homophorphically, using ι:
  - $[\![p(t_1, ..., t_n)]\!]^{I,\iota} = \begin{cases} 0 & \text{if } I(p)([\![t_1]\!]^{I,\iota}, ..., [\![t_n]\!]^{I,\iota}) = \bot \\ 1 & \text{if } I(p)([\![t_1]\!]^{I,\iota}, ..., [\![t_n]\!]^{I,\iota}) = \top \end{cases}$
- The conjunction $\wedge$ (resp. the disjunction $\vee$) *returns* 1 iff both (resp. at least one of the) sub-formulaes are (is) equal to 1:
  - $[\![\phi_1 \wedge \phi_2]\!]^{I,\iota} = min([\![\phi_1]\!]^{I,\iota}, [\![\phi_2]\!]^{I,\iota})$

  - $[\![\phi_1 \vee \phi_2]\!]^{I,\iota} = max([\![\phi_1]\!]^{I,\iota}, [\![\phi_2]\!]^{I,\iota})$

- The implication $\Rightarrow$ returns 1 iff the left sub-formula is evaluated to 0[2] or the right one is evaluted to 1:

  - $[\![\phi_1 \Rightarrow \phi_2]\!]^{I,\iota} = max(1 - [\![\phi_1]\!]^{I,\iota}, [\![\phi_2]\!]^{I,\iota})$

- The equivalence symbol $\Leftrightarrow$ *checks* that the two sub-formulae behave similarly:

  - $[\![\phi_1 \Leftrightarrow \phi_2]\!]^{I,\iota} = 1 - |[\![\phi_1]\!]^{I,\iota} - [\![\phi_2]\!]^{I,\iota}|$

- The negation simply switches an evaluation from 0 to 1, and the other way around:

  - $[\![\neg\phi]\!]^{I,\iota} = 1 - [\![\phi]\!]^{I,\iota}$

- The universal quantification $\forall$ (resp. existential quantification $\exists$) *returns* 1 iff the subformula is evaluated to 1 for any (resp. at least one) extension of the valuation wrt the quantified variable:

  - $[\![\forall x, \phi]\!]^{I,\iota} = \min_{u \in U_{\mathfrak{M}}} [\![\phi]\!]^{I,\iota[x \mapsto u]}$
  - $[\![\exists x, \phi]\!]^{I,\iota} = \max_{u \in U_{\mathfrak{M}}} [\![\phi]\!]^{I,\iota[x \mapsto u]}$

One does not need to be writing his or her doctoral thesis to know that logic, broadly speaking, revolves around the notion of truth. The truthness of a formula, called validity, is formally defined using the following notions.

Definition 1.35. A formula $\phi \in F_\Sigma(\mathcal{X})$ is satisfiable if and only if there exists a Σ-structure $\mathfrak{M} = (U_{\mathfrak{M}}, I)$ and a valuation $\iota : \mathcal{X} \to U_{\mathfrak{M}}$ such that $[\![\phi]\!]^{I,\iota} = 1$, which is written $\mathfrak{M}, \iota \models_I \phi$.

Definition 1.36. A formula $\phi \in F_\Sigma(\mathcal{X})$ is valid in $\mathfrak{M}$ iff $[\![\phi]\!]^{I,\iota} = 1$, for all valuations $\iota : \mathcal{X} \to U_{\mathfrak{M}}$. This is denoted as $\mathfrak{M} \models_I \phi$, and $\mathfrak{M}$ is called a **model** of ϕ.

Definition 1.37. A formula $\phi \in F_\Sigma(\mathcal{X})$ is valid in general iff $\mathfrak{M} \models_I \phi$ for all Σ-structures $\mathfrak{M}$. This is written $\models \phi$.

The notion of validity is also used to define that of logical consequence.

Definition 1.38. Let ϕ_1 and ϕ_2 be two formulae in $F_\Sigma(\mathcal{X})$. $\phi_1 \in F_\Sigma(\mathcal{X})$ **entails**, or implies, iff, for all Σ-structures $\mathfrak{M} = (U_{\mathfrak{M}}, I)$,

$$\mathfrak{M} \models_I \phi_1 \text{ implies that } \mathfrak{M} \models_I \phi_2$$

In that setting, ϕ_2 is said to be a **semantic consequence**, or logical implication, of ϕ_1, which is written $\phi_1 \models \phi_2$.

Definition 1.39. Formulae ϕ_1 and ϕ_2, both in $F_\Sigma(\mathcal{X})$, are **equivalent** to each other iff $\phi_1 \models \phi_2$ and $\phi_2 \models \phi_1$. This is denoted as $\phi_1 \equiv \phi_2$.

[2] Intuitively, if the precondition is false, the rest does not matter.

Example 1.40. The following, well-known equivalences can be verified using the formulae of Definition 1.34.

- $\phi \wedge \top \equiv \phi$; $\phi \wedge \bot \equiv \bot$; $\phi \vee \top \equiv \top$; $\phi \vee \bot \equiv \phi$
- $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
- $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$
- $\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \vee \neg\phi_2$
- $\neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$
  - These last two rules are known as the De Morgan's laws
- $\phi_1 \vee (\phi_2 \wedge \phi_3) \equiv (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$
- $\phi_1 \wedge (\phi_2 \vee \phi_3) \equiv (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$
  - These last two rules are the distributivity of $\vee$ over $\wedge$, and $\wedge$ over $\vee$, respectefully
- $\neg\neg\phi \equiv \phi$
- $\neg(\forall x, \phi) \equiv \exists x, \neg\phi$
- $\neg(\exists x, \phi) \equiv \forall x, \neg\phi$

Given a signature and an interpretative structure, the set of sentences that are satisfied by that framework is called a theory.

Definition 1.41. The **first-order theory** of a Σ-structure $\mathfrak{M} = (U_{\mathfrak{M}}, I)$ is defined as $Th(\mathfrak{M}) = \{\phi \in F_\Sigma(\mathcal{X}) \mid FV(\phi) = \emptyset \text{ and } \mathfrak{M} \models_I \phi\}$

Notation 1.42. In the rest of this document, $\mathfrak{M} \models_I \phi$ will simply be written $\mathfrak{M} \models \phi$.

Many first-order theories are studied and used, such as Peano arithmetic, Presburger arithmic, equality and so on. Datalog, which will be presented in Section 2, is also formalized as a first-order theory.

1.3 Normal forms

Like many other mathematical structures (e.g. matrices), first-order logic formulae are most efficiently used when in some so-called normal forms. One of them, the Horn clauses, are a key concept of Datalog. The following definitions build up to them, starting with a normal form that gathers all the quantifications at the beginning of a new, quantifier-free formula.

Definition 1.43. Any formula ϕ can be converted into a (semantically) equivalent formula in **Prenex Normal Form**. The prenex formula is of the form $\square_1 x_1 ... \square_n x_n \psi$, with $\square_i \in \{\forall, \exists\}$ for i, and ψ is quantifier-free. In that setting, $\square_1 x_1 ... \square_n x_n$ is called the quantifier prefix, and ψ the matrix. This transformation is denoted as $\Rightarrow_P^*$.

The next transformation, called skolemization, erases the existential quantifications in the quantifier prefix.

Definition 1.44. Skolemization converts any prenex formula ϕ into an equally satisfiable skolem formula $\forall x_1 ... \forall x_n \psi$. The transformation embeds every (previously) existentially quantified variable into an explicit choice function that depends on every previously quantified variable. More formally, it applies the following transformation until all existential quantifications have been eliminated:

$$\forall x_1 ... \forall x_n \exists y \phi \Rightarrow_{Sk} \forall x_1 ... \forall x_n [y \mapsto f(x_1, ..., x_n)] \psi$$

The skolem formulae can then be transformed into conjunctive normal form.

Definition 1.45. A formula in **Conjunctive Normal Form** (or Clausal Normal Formal, CNF in both cases) is a conjunction of clauses, i.e. a formula of the form

$$\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} L_{ij}$$

where all L_{ij} are atomic literals. Any skolem formula can be transformed into a CNF *via* a procedure found in [Russell and Norvig, 2009]. Basically, it gets rid of any unnecessary negation using some of the equivalences shown in Example 1.40, puts the conjunctions *above* the disjunctions using the equivalence $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, and then simply erases the quantifications.

We finally introduce a subclass of CNF, which is used extensively in Datalog, and thus in Section 2.

Definition 1.46. A clause with at most one positive literal is called a **Horn clause**.

Definition 1.47. A CNF first-order logic formula $\phi = \bigwedge_{i=1}^{m} C_i$, where every C_i is a Horn clause, is called a **Horn formula**.

Remark 1.48. Using equivalences in Example 1.40, we can show that $\neg L_1 \vee ... \vee \neg L_n \vee L_p$ is equivalent to $(L_1 \wedge ... \wedge L_n) \Rightarrow L_p$. A Horn clause can then be seen as a list of preconditions leading to (at most) one result. Lifting this to full formulae, a Horn formula is then akin to a list (conjunction) of rules which can be interpretated as "if L_1 and ... and L_n are true, then so is L_p".

1.4 Inference

We have seen first-order logic as a way to formalize and manipulate statements, but we also need to be able to relate them.

Definition 1.49. An **inference system** is a set of judgements, or rules describing how to deduce new facts from a previously established set of formulae. They are presented as

$$\frac{J_1 \quad \dots \quad J_n}{J_{n+1}} \, R$$

where J_{n+1} is the (syntactical) consequence of the hypotheses J_1 to J_n, and R is the label, or name, of the rule. This can also be written $J_1, ..., J_n \vdash_R J_{n+1}$. If $n = 0$, i.e. there is no required hypothesis, the judgement is called an **axiom**.

Definition 1.50. The hypotheses in an inference can themselves be the result of another inference, and so on. In that sense, the inference rules can be applied iteratively, which is called a **derivation**. When F can be deduced from the set of base hypotheses Δ using such a derivation in the inference system $\mathcal{I}$, it is denoted as $\Delta \vdash_{\mathcal{I}} F$.

These inference rules are purely syntactical, in the sense that they need to be related to the semantics of the used language. We now introduce the most important concepts.

Definition 1.51. Let $\mathcal{I}$ be an inference system for a language $\mathcal{L}$. If, for any set of closed formulae $\Delta \subseteq SEN_{\mathcal{L}}$ and $F \in SEN_{\mathcal{L}}$:

- $\Delta \vdash_{\mathcal{I}} F$ implies $\Delta \models F$, then $\mathcal{I}$ is **sound**;
- $\Delta \models F$ implies $\Delta \vdash_{\mathcal{I}} F$, then $\mathcal{I}$ is **complete**;
- $\Delta \not\models F$ implies $\Delta \cup \{F\} \vdash_{\mathcal{I}} \bot$, then $\mathcal{I}$ is **refutationally complete**.

In other words, an inference system is complete if there exists a syntactical deduction for every semantic implication (the syntax *covers* the semantics), sound if a syntactical deduction corresponds to a semantic implication (the syntax does not *step out* of the semantics), and refutationally complete if adding as an axiom a statement that is not a semantic implication can lead to the deduction of $\bot$, i.e. the *false* statement.

Remark 1.52. In practice, refutational completeness is used in deduction systems by first adding the negation of the statement one wants to prove to the set of hypotheses, and then showing that the *false* statement can be derived.

We skim over the historic and fundamental examples of inference systems, e.g. natural deduction, Hilbert or sequents, to focus on a specific, more computation-oriented family of techniques. The **resolution** based inference techniques work with clausal formulae, as illustrated in the following example.

Example 1.53. The **Binary Resolution Inference** system consists of two rules:

$$\frac{A \vee C \quad B \vee \neg C'}{\sigma(A \vee B)} \text{ Binary resolution}$$

where σ is the most general unifier (mgu) of C and C', i.e. the smallest substitution such that $\sigma(C) = \sigma(C')$. Intuitively, this rule looks for a substitution (which can be seen as constraints) such that C and C' match, meaning that C and $\neg C'$ become incompatible. Given the hypotheses, at least one of the left compotents, also subjected to the substitution, must be true.

The second rule is

$$\frac{A \vee B \vee C}{\sigma(A \vee B)} \text{ Factoring}$$

where σ is the mgu of B and C. In other words, this rule tries to *collapse* two atoms in a disjunction.

As previously stated, resolution is not an inference system in itself, but rather a family of similar techniques [Bachmair et al., 2001]. In Section 2.2, we will revisit hyperresolution, introduced in [Robinson, 1974].

Definition 1.54. The **hyperresolution rule** is formulated as

$$\frac{A_1 \vee C_1 \quad \dots \quad A_n \vee C_n \quad B \vee \neg C_1' \vee \dots \vee \neg C_n'}{\sigma(A_1 \vee ... \vee A_n \vee B)}$$

where σ is the mgu of C_i and C_i', for every $i \in [1, n]$.

Remark 1.55. Hyperresolution is a generalization, or rather iteration, of the binary resolution rule introduced in Example 1.53. Its implementation can be augmented with multiple heuristics, regarding ordering and selection.

Theorem 1.56. Hyperresolution is sound and refutationally complete.

Proof. See [Bachmair et al., 2001]. □

Chapter 2

Datalog

> Rien ne dépasse la beauté simple et froide de la logique. Si Galilée pouvait prétendre que la Nature est un livre écrit en langage mathématique, c'est qu'il n'existe rien de plus élégant qu'un système parfaitement ordonné, où chaque conséquence est le fruit d'une clause, où chaque élément est imbriqué dans un tout plus grand que la somme de ses parties.
>
> Emmanuel Denise, Canard PC 396

Datalog is a simple and declarative language, based on first-order logic and tuned to data-centric applications, usually described as "Prolog without function symbols" [Liu, 1999]. The author of [Greenman, 2017] tracks its origins in the sixties and seventies, when mathematical logic was first considered as a lens through which databases could be seen.

More concretely, relational algebra is proposed as the fundational bases for the relational model of databases in 1970 [Codd, 1970], the first Prolog interpreter is developed in 1973 by Colmenaur and his students, and a programming language semantics of predicate logic, i.e. of Datalog, is introduced in 1976 [Van Emden and Kowalski, 1976].

Originally designed as a powerful query language on databases, Datalog has since then gained interest thanks to domain-specific extensions [Abiteboul et al., 1995, Ramakrishnan and Ullman, 1995]. The introduction of [Benzaken et al., 2017a] gives a comprehensive list of languages built upon Datalog [Lu and Cleary, 1999, Loo et al., 2005, Grumbach and Wang, 2010, Calì et al., 2009, Seo et al., 2013, Aref et al., 2015] and applications, in both academic [DeTreville, 2002, Whaley et al., 2005, Hellerstein, 2010, Huang et al., 2011] and industrial [Chin et al., 2015, Gottlob et al., 2004, log, 2020, dat, 2020, sem, 2020] settings.

As a first approximation, Datalog is a fragment of Prolog without function symbols. A program is then a set of Horn clauses. Some of these clauses have no *tail*, or precondition, and constitute a first set of facts. The semantics of a program is this set of initial facts as well as those that can be deduced in any number of steps using the other clauses, called rules. A key feature of Datalog is recursivity, which makes it possible to compute transitive closures, e.g. accessibility in graphs, in a simpler and more complete way than other query languages, such as SQL, XPath and SPARQL.

In contrast to Prolog, the evaluation mechanism of Datalog follows a bottom-up strategy which guarantees termination even in the presence of recursive rules [Abiteboul et al., 1995].

The key idea is that, without function symbols, the set of derivable facts is always finite.

Section 2.1 and 2.2 formalize the syntax and semantics of Datalog, then Sections 2.3 and 2.4 discuss how Datalog can be augmented with negation and runtime computations.

2.1 Syntax

We first present the rules and constraints upon which Datalog programs are built.

2.1.1 Building blocks

As previously stated, a Datalog program is a set of Horn clauses (Definition 1.46), split into facts and rules. We build up the syntax, starting with the different kinds of symbols.

Definition 2.1. Datalog symbols are either arity bound predicates, constants or variables. We fix $\mathcal{P}$ as the set of predicates together with an arity function $ar : \mathcal{P} \to \mathbb{N}$, $\mathcal{C}$ as the set of constants and $\mathcal{V}$ as the set of variables.

These symbols are used to build the expressions, i.e. terms, atoms, clauses and programs.

Definition 2.2. A **term** t is either a constant or a variable.

$$t ::= x \mid c, \text{ where } x \in \mathcal{V}, c \in \mathcal{C}$$

Definition 2.3. Let p be a predicate and $\vec{t}$ a term sequence with $|\vec{t}| = ar(p)$. An **atom** A is an expression of the form

$$A ::= p(\vec{t})$$

Notation 2.4. The terms $t_1, t_2, ..., t_n$ are the arguments of the atom. The predicate of an atom is accessed with function sym.

Definition 2.5. A **clause** C is defined as

$$C ::= A_0 \leftarrow A_1, ..., A_m.$$

Notation 2.6. The atom A_0 is called the head of C, whereas the atom list $A_1, ..., A_m$ is its body. In the rest of this document, the writing convention will be to use H for the head of a clause, and $B_1, ..., B_m$ for the atoms of the body.

Definition 2.7. As stated in Remark 1.48, a Horn clause can be understood as "if B_1 and B_2 ... and B_m, then H". In that spirit, when $m = 0$, i.e. $C \equiv H \leftarrow$ (or simply H), the clause represents a **fact**. Otherwise, when $m \geq 1$, a clause is called a **rule**.

Definition 2.8. A **program** P is a finite set of clauses.

$$P ::= C_0, ..., C_k, \text{ where the commas denote conjunction.}$$

$path(X, Y) \leftarrow edge(X, Y).$
$path(X, Y) \leftarrow path(X, Z),\ \ edge(Z, Y).$

$edge(1, 3).$
$edge(2, 1).$
$edge(4, 2).$
$edge(2, 4).$

Figure 2.1: Directed graph connectivity in Datalog

Example 2.9. Figure 2.1 shows a textbook Datalog program that computes the (non-empty) paths in a directed graph. The first two lines are rules, whereas the last four are facts.

The Datalog programs are however not built entirely *à la carte*, as they must enforce two contraints.

2.1.2 Extensional vs. intensional predicates

Definition 2.10. In any Datalog program, the involved predicates must be split into two classes: **extensional** (or base) predicates, that are only defined using facts, and **intensional** (or derived) predicates, that are defined only *via* rules.

Corollary 2.11. In any Datalog program, a predicate can not appear as the head of both a rule and a fact.

Example 2.12. In the program of Figure 2.1, the `path` predicate is intensional (defined by the two rules), whereas `edge` is extensional (defined by the four facts), and Corollary 2.11 is respected.

Remark 2.13. From a database standpoint, the extensional predicates correspond to the relations actually stored in the database, whereas the intensional predicates are *virtual, computed relations*, usually called *views*. In that setting, the facts of a program P form the **extensional database**, written $edb(P)$, the rules are the **intensional database** $idb(P)$, and the program schema is $sch(P) = edb(P) \cup idb(P)$.

Although Datalog programs are formalized as a mix of rules and facts in this section, the (usually) huge preponderance of facts over rules makes it convenient and more efficient to view them as two separate components. As discussed in Chapter 3, this is actually done in the aforementioned Coq formalization of Datalog [Dumbrava, 2016].

Notation 2.14. In the rest of this document, the considered programs will be clearly identified and the set of base facts shall be denoted as simply EDB, rather than $edb(P)$.

Remark 2.15. Surprisingly, this constraint does not mitigate the expressivity of Datalog. Assume an invalid program, where a predicate p is used both in an intensional and extensional manner. We can introduce a purely extensional predicate p_{EDB}, which contains the same facts as p, and a rule $p(X) \leftarrow p_{EDB}(x)$. Then, we can remove p from the EDB, making it a purely intensional predicate, while preserving the originally intended semantics.

2.1.3 Program safety

On the other hand, the second constraint does limit the expressivity of Datalog. To formalize it, we first need the following definition, which will also be used extensively when introducing the semantics of Datalog.

Definition 2.16. The variable-free expressions of Datalog are called **ground expressions**. The sets of ground atoms, clauses and programs are written $\overline{A}, \overline{C}$ and $\overline{P}$, respectively.

Definition 2.17. A clause is safe if all the variables in its head also appear in its body.

Remark 2.18. Since it has no body, **a fact is safe** iff it is ground.

Definition 2.19. (Program safety condition) A program P is safe if all the clauses it contains are safe.

Example 2.20. The program in Figure 2.1 is safe. Note that the last four clauses, or facts, are an illustration of Remark 2.18.

Example 2.21. The program of Figure 2.2a is invalid in Datalog, as variable X appears in the head of the only rule but not the body (note that the safety constraint more generally forbids rules with empty bodies). On the other hand, the program of Figure 2.2b, where $element(X)$ is an extensional predicate that contains every constant of the EDB, is a hacky but valid program. Finally, note that in practice, equality is seen as a primitive predicate (see Section 2.4).

$equal(X, X) \leftarrow .$

(a) Invalid definition

$equal(X, X) \leftarrow element(X).$

(b) Valid definition

Figure 2.2: Expressing equality in Datalog

Remark 2.22. The safety conditions may seem arbitrary at first, but they **bound the semantics** of any given Datalog program and guarantee the termination of the bottom-up evaluation strategy. This is discussed more in-depth hereafter.

Example 2.23. Datalog can be used for many access control policies or rule-based processes. For example, [Dougherty et al., 2006] presents a Datalog formalization of the access to conference review scores. As an alternative *real life situation*, Figure 2.3 translates into Datalog a simplified version of the access to the parisian MK2 Bibliothèque movie theater.

$can\_watch(P, M, D) \leftarrow showing(D, M), cost(P, D, C), pays\_for(F, P, C).$

$cost(P, D, 4.90) \leftarrow child(P), any(D).$
$cost(P, D, 4.90) \leftarrow young(P), week(D).$
$cost(P, D, 7.90) \leftarrow young(P), weekend(D).$
$cost(P, D, 7.90) \leftarrow student(P), any(D).$

Figure 2.3: Buying a ticket at the MK2 Bibliothèque

Although the semantics of Datalog has not been formally introduced at this point of the document, the sharp-eyed reader will probably infer the informal meaning of the rules. The

first one states that a person P can go see a movie M on day D if

- M is shown on D
- seeing a movie on D will cost C to P
- a person F (which in practice can be P him/herself, or maybe a friend) pays C for P

The other rules encode a fragment of the pricing table[1]. Note that the program safety condition in enforced using the *any* predicate, which contains every day constant. This trick can be used because there is only a finite number of possible values, i.e. seven days.

2.2 Semantics

We introduce two semantics for Datalog: the Minimal Model Semantics, which roughly views Datalog as a first-order logic theory, and the Fixpoint Semantics, which is more akin to an execution engine. These two semantics stem from the key result of fixpoint theory, called the Knaster-Tarski theorem [Tarski, 1955].

Theorem 2.24. (Knaster-Tarski theorem) Let us assume a complete lattice $\langle \mathcal{L}, \leq \rangle$ and an operator f on $\mathcal{L}$. If f is monotonic, then it has a least fixpoint, denoted as $lfp(f)$.

The original proof establishes that pre-fixpoint are closed by min, meaning that the least fixpoint is the inf over all pre-fixpoints. This is the theoretical background of the Minimal Model Semantics detailed in Section 2.2.1.

If the lattice is finite, this original proof provides an algorithm to compute the least fixpoint. However, enumerating every pre-fixpoints is obviously not always feasible in practice. Another proof, attributed to Kleene, provides a more efficient algorithm. This algorithm iterates the f operator, starting from the bottom of the lattice, until a fixpoint is reached. This fixpoint is shown to be the least fixpoint. This second proof is the theoretical background of the Fixpoint Semantics detailed in Section 2.2.2.

Remark 2.25. Calling only the second semantics the (and not "a") *fixpoint semantics* may be misleading, as both are about fixpoints.

As expected given the fact that these two semantics are inspired by two different proofs of the same theorem, they are equivalent, as shown in [Van Emden and Kowalski, 1976].

2.2.1 Minimal Model Semantics

As previously seen, a Datalog program P is a set of Horn clauses, meaning that it can easily be translated into an actual first-order logic formula P^* using the following rules:

Atom An atom A can simply be represented as a first-order logic formula A^*

[1] At the time of writing, the full pricing specification of the MK2 Bibliothèque can be found at https://www.mk2.com/sites/default/files/grille_tarifaire_bibliotheque2.pdf. A complete Datalog implementation is of course left as an exercice to the reader.

Clause A clause $C \equiv H \leftarrow B_1, ..., B_m$ can then be translated into

$$C^* \equiv \forall X_1, ..., \forall X_q((\exists X_{q+1} ... \exists X_l, (B_1 \wedge ... \wedge B_m)) \Rightarrow H)$$

with $\{X_1, ..., X_q\} = VAR(H)$
and $\{X_{q+1}, ..., X_l\} = \bigcup_{k \in [1,m]} VAR(B_k) \backslash VAR(H)$

In other words, given an instantiation $[X_1 \mapsto c_1, ..., X_m \mapsto c_m]$ of the variables in the clause's head, $H(c_1, ..., c_m)$ can be deduced if the rest of the clause's variables can be instantiated so that the resulting ground atoms are all true.

Program As a program P is a set of clauses, P^* is simply $\bigwedge_{C \in P} C^*$

Example 2.26. (Graph Transitive Closure) Using these rules to translate the program of Figure 2.1 returns the following formula, where the first line contains the facts, and the other two correspond to the two rules of the program.

$$\begin{aligned} & e(1,3) \wedge e(2,1) \wedge e(4,2) \wedge e(2,4) \\ \wedge\ & (\forall X, \forall Y, (edge(X,Y) \Rightarrow path(X,Y))) \\ \wedge\ & (\forall X, \forall Y, (\exists Z, (edge(X,Z) \wedge path(Z,Y))) \Rightarrow path(X,Y)) \end{aligned}$$

Now that Datalog programs can be translated to first-order logic, we can define their semantics in this framework. To do so, we provide the interpretation (cf. Definition 1.27) of the fragment of FOL corresponding to the translation.

Definition 2.27. Let us assume a program P and the signature $\Sigma = (\mathcal{C}, \mathcal{P}, ar)$. As in Section 2.1.1, $\mathcal{C}$ is the set of program constants, and ar the arity function for the set of predicates $\mathcal{P}$. A **Σ-structure** $\mathcal{I}$ comes in the form $\mathcal{I} = (U, I : \Sigma' \to U \cup \{\top, \bot\})$, where $\Sigma' = \mathcal{C} \cup \mathcal{P}$. The codomain of interpretation I is the union of the non-empty universe U and the boolean set, the former being used for the interpretation of constants and the latter for atoms. More formally,

- for every $c \in \mathcal{C}, I(c) \in U$
- for every $p \in \mathcal{P}, I(p)$ is a mapping $U^{ar(p)} \to \{\top, \bot\}$. In other words, the interpretation expects a number of arguments as specified by the arity function for the given predicate and returns a boolean evaluation.

Definition 2.28. We can now define the actual interpretation function. Assuming a signature Σ, a set of variables $\mathcal{V}$ and a valuation $\iota : \mathcal{V} \to U$, the **interpretation** $[\![e]\!]^{I,\iota}$ of a Datalog expression e is defined by structural induction on e using the following rules.

- $[\![x]\!]^{I,\iota} = \iota(x)$
- $[\![c]\!]^{I,\iota} = I(c)$
- $[\![p(t_1, ..., t_n)]\!]^{I,\iota} = I(p)([\![t_1]\!]^{I,\iota}, ..., [\![t_n]\!]^{I,\iota}) = \begin{cases} \top & \text{if } I(p)([\![t_1]\!]^{I,\iota}, ..., [\![t_n]\!]^{I,\iota}) = \top \\ \bot & \text{otherwise} \end{cases}$
- $[\![H \leftarrow B_1, ..., B_n]\!]^{I,\iota} = \begin{cases} \top & \text{if } \exists i \in [1,n], [\![B_i]\!]^{I,\iota} = \bot \vee [\![H]\!]^{I,\iota} = \top \\ \bot & \text{otherwise} \end{cases}$

- $[\![H \leftarrow B_1, ..., B_n]\!]^I = \begin{cases} \top & \text{if } \forall \iota, [\![H \leftarrow B_1, ..., B_n]\!]^{I,\iota} = \top \\ \bot & \text{otherwise} \end{cases}$

Definition 2.29. An **interpretation** I is a **model** for a clause C if the latter is evaluated to $\top$ wrt the former, i.e. $[\![C]\!]^I = \top$. Such an interpretation is also a model for a program P if it is for every clause of the program, i.e. $\forall C \in P, [\![C]\!]^I = \top$.

We now give a FOL view of the deduction of new facts in Datalog. To do so, we rely on Herbrand Semantics (cf. Section 1.2).

Definition 2.30. A fact F is a **logical consequence** of a program P, written $P \models F$, iff any interpretation I satisfying P also satisfies F, i.e. $I \models P$ implies $I \models F$.

Notation 2.31. The set of **all logical consequences of a program** P is denoted as $cons(P)$.

Definition 2.32. We express the **logical consequences of a Datalog program** P using the Herbrand Semantics of P, the building blocks of which are:

- as Herbrand Universe, the set of all program constants, written $adom(P)$.
- as Herbrand Base, the set of all ground atoms that can be built from predicates (in $\mathcal{P}$) and constants in $adom(P)$, written $\mathbb{B}_P$.
- as Herbrand Interpretation, $I_\mathcal{H}$ (cf. Definition 1.29), i.e. the symbols are their own interpretation.

Remark 2.33. As seen in the definition of the Herbrand Universe, the only constants which need to be considered are those that actually appear in the studied program. This stems from the safety condition (see Section 2.1.3) and the absence of terms, meaning that Datalog programs only *pass around* values rather than introducing or computing new ones, as can be done in Prolog for example.

Definition 2.34. In that setting, **the grounding of a program** P is defined as a valuation (cf. Definition 1.30) $\iota : BV(P) \to adom(P)$ as

$$\iota(P) = \bigcup_{C \in P} \iota(C)$$

with $\iota(p_0(\vec{t_0}) \leftarrow p_1(\vec{t_1}), ..., p_m(\vec{t_m})) = p_0(\iota(\vec{t_0})) \leftarrow p_1(\iota(\vec{t_1})), ..., p_m(\iota(\vec{t_m}))$

Definition 2.35. Applying this definition to the **satisfaction of a clause wrt a Herbrand interpretation** $I_\mathcal{H}$,

$$I_\mathcal{H} \models \iota(C) \text{ iff } \{p_1(\iota(\vec{t_1})), ..., p_m(\iota(\vec{t_m}))\} \subseteq I_\mathcal{H} \text{ implies that } p_0(\iota(\vec{t_0})) \in I_\mathcal{H}$$

We can now define, as a special case of Definition 1.36, the notion of Herbrand model.

Definition 2.36. A Herbrand interpretation $I_\mathcal{H}$ is a **Herbrand model** of a program P iff $I_\mathcal{H} \models \iota(P)$, for all $\iota : BV(P) \to adom(P)$

Example 2.37. Let P be the program $\{p(a), q(a), r(X) \leftarrow p(X)\}$. The set of predicates $\mathcal{P}$ is $\{p, q, r\}$, and that of constants, $adom(P) = \{a, b\}$. They can be used to build the set of

ground atoms $\mathbb{B}_P = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$.

Now consider the following Herbrand interpretations:

- $I_1 = \emptyset$
- $I_2 = \{p(a), q(a)\}$
- $I_3 = \{p(a), q(a), r(a)\}$
- $I_4 = \{p(a), q(a), r(a), p(b), r(b)\}$
- $I_5 = \{p(a), q(a), r(a), q(b)\}$
- $I_6 = \mathbb{B}_P$

I_1 is not a Herbrand model of P as it does not even contain the facts of the program. I_2 does have them, but the $r(X) \leftarrow p(X)$ rule is not enforced, as it contains $p(a)$ and not $r(a)$. On the other hand, I_3, I_4, I_5 and I_6 are Herbrand models of P.

Theorem 2.38. If M_1 and M_2 are Herbrand models of a definite program P, then $M_1 \cap M_2$ is also a model of P. This theorem is called the **Model intersection property**.

Proof. Let C be a Horn clause in P and ι a valuation. C is either a fact or a rule. In the first case, M_1 and M_2 must both contain $\iota(C)$ to be models of P, which means that $M_1 \cap M_2$ does too.

If $C \equiv p_0(\vec{t_0}) \leftarrow p_1(\vec{t_1}), ..., p_m(\vec{t_m})$ is a rule, then M_1 and M_2 both either contain $\iota(p_0(\vec{t_0}))$, or do not contain a $\iota(p_i(\vec{t_i}))$ with $t_i \in [1, m]$. If M_1 and M_2 indeed both have $\iota(p_0(\vec{t_0}))$, then so does $M_1 \cap M_2$. If M_1 or M_2 does not contain a $\iota(p_i(\vec{t_i}))$, then $M_1 \cap M_2$ does not either. In both cases, $M_1 \cap M_2$ satisfies C. □

Definition 2.39. Using set inclusion as a partial order, a Herbrand model is called **minimal** if none of its proper subsets are also models.

Definition 2.40. As a consequence of Theorem 2.38, any program P has a **unique minimal model** $M(P)$, the intersection of all its Herbrand models.

Example 2.41. In Example 2.37, I_3 is the minimal model of P. Although all models of the program are not listed, the reader can note that $I_3 = I_4 \cap I_5 \cap I_6$.

The unique minimal model is the **intended semantics** of Datalog. In other words, given a Datalog program P, $cons(P) = M(P)$.

Remark 2.42. This choice, as explained in Chapter 12.2 of [Abiteboul et al., 1995], is a consequence of a philosophical hypothesis fueling database theory, called the **closed world assumption**. It basically states that dabases should be considered to be complete, although they obviously do not contain every fact about the world in practice. In that setting, any fact which can not be derived (or proved) from a program should be considered false. The minimal model semantics enforces this philosophy.

2.2.2 Fixpoint Semantics

We now give an alternative, more computation-oriented but equivalent semantics of Datalog. This semantics being based upon fixpoint theory, we first recall its basics.

Definition 2.43. A **complete lattice** is a pair $\langle \mathcal{L}, \leq \rangle$, where $\mathcal{L}$ is an ordered set wrt to the partial order $\leq$ and any set $A \subseteq \mathcal{L}$ has a greatest lower bound $\bigcap A$, and a lowest upper bound $\bigcup A$.

Definition 2.44. Given a set $\mathcal{L}$ equipped with a partial order $\leq$, an operator $f : \mathcal{L} \to \mathcal{L}$

- is **monotonic**, if $x_1 \leq x_2$ implies $f(x_1) \leq f(x_2)$ for all $x_1, x_2 \in \mathcal{L}$, i.e. f preserves the partial order $\leq$
- has a **pre-fixed point** p if $f(p) \leq p$
- has a **fixpoint** x if $f(x) = p$

We can now apply the Knaster-Tarski theorem (see Theorem 2.24) to the interpretation of Datalog. The partially ordered set here is the set of Herbrand interpretations $\mathcal{P}(\mathbb{B}_P)$, i.e. the different combinations of ground atoms. The operator is introduced in the following definition.

Definition 2.45. Given a program P, the **immediate consequence operator** T_P works on interpretations, i.e. $T_P : \mathcal{P}(\mathbb{B}_P) \to \mathcal{P}(\mathbb{B}_P)$, and is defined as

$$T_P(I) = I \cup \{head(\iota(C)) \mid C \in P \wedge \iota \text{ a grounding of } P \wedge body(\iota(C)) \subseteq I\}$$

In other words, T_P preserves the ground atoms in its argument ($F \in I$), and adds the heads of clauses in the program instantiated with valuations such that the ground atoms in the tail are all in the argument (right side of the $\vee$). Any ground atom F *produced* by the operator is called an immediate consequence of the program.

Lemma 2.46. T_P is monotonic.

Proof. Let us assume I_1 and $I_2 \subseteq \mathbb{B}_P$ such that $I_1 \subseteq I_2$, and $F \in T_P(I_1)$. Either $F \in I_1$, or there exists a $C \in P$ and a valuation ι such that $body(\iota(C)) \subseteq I_1$. In both cases, since I_1 is a subset of I_2, the former can be replaced by the latter in the assertion, showing that F is also in $T_P(I_2)$. □

Theorem 2.47. T_P has a least fixpoint, $lfp(T_P)$. In practice, it is computed by iterating T_P, starting with the minimal, i.e. empty interpretation.

Proof. Combination of Theorem 2.24 and Lemma 2.46. □

Definition 2.48. The **powers of the immediate consequence operator** are defined as

$$\begin{cases} T_P \uparrow 0 = \emptyset \\ T_P \uparrow (n+1) = T_P(T_P \uparrow n) \end{cases}$$

As previously stated, there exists some $\omega \in \mathbb{N}$ such that $T_P \uparrow \omega = \bigcup_{n \geq 0} T_P \uparrow n = lfp(T_P)$

Example 2.49. Let P be the graph connectivity program from Example 2.9, then

- $T_P \uparrow 0 = \emptyset$
- $T_P \uparrow 1 = T_P(\emptyset) = \emptyset \ \cup \ \{edge(1,3), edge(2,1), edge(4,2), edge(2,4)\}$
- $T_P \uparrow 2 = T_P(\emptyset) = T_P \uparrow 1 \ \cup \ \{path(1,3), path(2,1), path(4,2), path(2,4)\}$
- $T_P \uparrow 3 = T_P(T_P \uparrow 2) = T_P \uparrow 2 \ \cup \ \{path(2,3), path(4,1), path(4,4), path(2,2)\}$
- $T_P \uparrow 4 = T_P \uparrow 3 \cup \{path(4,3)\}$
- $T_P \uparrow 5 = T_P(T_P \uparrow 4) = T_P \uparrow 4$.

It then appears that $T_P \uparrow \omega = T_P \uparrow 4 = lfp(T_P)$.

Lemma 2.50. Given a Datalog program P and a Herbrand Structure $\mathcal{H} = (U_\mathcal{H}, I)$, I is a pre-fixed point of T_P iff $I \models P$.

Proof.

- $\Rightarrow$ Let I be a pre-fixed point of T_P, i.e. $T_P(I) \subseteq I$, $H \leftarrow B_1, ..., B_n$ a clause in P and ι a valuation. If $\{\iota(B_1), ..., \iota(B_n)\} \subseteq I$ then $\iota(H) \in T_P(I)$ (second condition in the definition of T_P). Since $T_P(I) \subseteq I$, $\iota(H) \in I$, and thus $I \models \iota(H \leftarrow B_1, ..., B_n)$.
- $\Leftarrow$ Let $\overline{A}$ be a ground atom in $T_P(I)$, we need to show that $\overline{A} \in I$. Then, based on T_P's definition, $\overline{A}$ was either already present in I, or there exists a rule $H \leftarrow B_1, ..., B_n \in P$ and a valuation ι such that $\overline{A} = \iota(H)$ and $\{\iota(B_1), ..., \iota(B_n)\} \subseteq I$. Since $I \models P$, $\iota(H) = \overline{(}A) \in I$.

□

Remark 2.51. Since $T_P(I)$ *preserves* the elements of I, $I \subseteq T_P(I)$. With that in mind, Lemma 2.50 can be re-stated as I is a fixpoint of T_P iff $I \models P$.

We can now relate the fixpoint and minimal model semantics.

Theorem 2.52. The unique minimal Herbrand Model $M(P)$ of a Datalog program P is $lfp(P) = T_P \uparrow \omega$.

Proof. See [Van Emden and Kowalski, 1976]. □

We end this section with a more operational view on the fixpoint semantics. Indeed, concretely, the consequence operator is equivalent to the following inference rule, called the Elementary Production Principle [Ceri et al., 1989].

$$\frac{H \leftarrow B_1, ..., B_n \qquad \{F_1, ..., F_n\} \subseteq I \qquad \exists \sigma, \sigma(B_1) = F_1 \wedge ... \wedge \sigma(B_n) = F_n}{\sigma(H)} \ EPP$$

Remark 2.53. The EPP rule is an implementation of hyperresolution, (see Section 1.54).

Theorem 2.54. Given a Datalog program P, the semantics of the program $cons(P)$ can be obtained by iterating the EPP rule until a fixpoint is reached. This algorithm is indeed both sound and complete.

Proof. Corollary of the completeness and soundness properties of hyperresolution established in [Bachmair et al., 2001]. □

2.3 Adding and handling negation

So far, we have introduced the so-called standard Datalog. It can however be augmented with various features, most notably negation. As explained in Chapter 6, this thesis, and the Datalog optimizations it introduces, were inspired by a network verification tool, called Octant, which did not initially scale. These performance issues were brought by the use of negation, meaning that we need to get an idea of how it is added and handled.

As always, we first discuss the syntactic side, before moving to the so-called stratified semantics. This semantics of Datalog with negation serves as the theoretical foundation for the implementation of negative Datalog in [Dumbrava, 2016] and will provide a good intuition of what happens during the execution of Octant. For the sake of being exhaustive, we will finally overview some alternative semantics of this extension.

2.3.1 Syntax

The first step is, unsurprisingly, to add a negation unary operator to Datalog's vocabulary. We shall write it $\neg$, as illustrated by the following example.

Example 2.55. The program of Figure 2.4 is similar to the one shown in Figure 2.1, which defined graph connectivity, with the addition of a rule. This (third) rule defines $disjoint$, the predicate, or set, of pairs of edges which are not connected.

$$
\begin{array}{ll}
path(X,Y) & \leftarrow edge(X,Y). \\
path(X,Y) & \leftarrow path(X,Z), edge(Z,Y). \\
disjoint(X,Y) & \leftarrow \neg path(X,Y) \\
 & \\
edge(1,2). & \\
edge(2,1). & \\
edge(2,3). &
\end{array}
$$

Figure 2.4: Directed graph disconnectedness in Datalog augmented with negation

Integrating this operator into Datalog requires a slight extension of its formal syntax, as defined in Section 2.1. Concretely, we add a new layer on top of the atoms, called literals (see Definition 1.11), and adapts the notion of clause.

Definition 2.56. A **literal** L is either a positive or a negated atom:

$$L ::= A \mid \neg A$$

Definition 2.57. A **clause** C has a positive atom head and a body of literals:

$$C ::= A_0 \leftarrow L_1, ..., L_m$$

Notation 2.58. The set of positive and negative atoms in the body of a clause C are written $body^+(C)$ and $body^-(C)$, respectively.

Remark 2.59. Section 6.1 of [Dumbrava, 2016] discusses the adaptation of the safety condition (see our own Section 2.1.3) to this new setting, as the use of negation opens the door to considering and computing an infinity of facts. However, it is stated that the previous notion is sufficient when the domain of values is restricted to the constants appearing in the program's database. Since this is the approach taken by the associated development [Benzaken et al., 2017b], and then by us as well, we do not dwelve into these considerations in this document.

While the changes to the syntax are minor, defining a semantics for Datalog augmented with negation is more involved. We first focus on a semantics called stratified semantics, and then overview some alternative approaches.

2.3.2 Stratified Semantics

The presentation of the stratified semantics will itself be stratified, as we first present a semantics for Datalog programs which use negation in a very limited way. Then, we will see how the idea behind this semantics can be lifted to unrestriced programs, by introducing a new logical consequence operator and an associated new iteration, as well as the notion of stratification of Datalog programs.

2.3.2.1 Semipositive Datalog

We start with a restricted use of negation in Datalog, which provides a simple intuition that will then serve as the bedrock of a generalization to more generic programs.

Definition 2.60. A **semipositive** Datalog program is a program where negation is only applied to atoms built with extensional predicates (see Section 2.1.2), i.e. predicates only defined by the EDB.

Example 2.61. The program shown in Figure 2.4 is not semipositive, since a negation is applied to *path*, which is an intensional predicate.

This restriced setting is simply dealt with, as such negated atoms can be replaced by their complement w.r.t. the program's Herbrand base, i.e. the set of relevant facts that can be built using constant appearing in the EDB. Then, the semantics of standard Datalog can be reused. The following example illustrates this mechanism.

$P(X,Y) \leftarrow R(X,Y),\ \neg R(Y,X).$

$R(1,2).$
$R(2,2).$

(a) A semipositive program...

$P(X,Y) \leftarrow R(X,Y), nR(Y,X).$

$nR(1,1).$
$nR(2,1).$
$R(1,2).$
$R(2,2).$

(b) and an equivalent standard Datalog program

Figure 2.5: Interpretation of semipositive Datalog programs

Example 2.62. Figure 2.5 shows a semipositive Datalog program P. Considering the set of constants that appear in the EDB, the only ground atoms about R in $\mathbb{B}_P$, the program's Herbrand base, are $R(1,1)$, $R(1,2)$, $R(2,1)$ and $R(2,2)$.

This program is then equivalent to Figure 2.5b, where a new predicate nR, is introduced to replace the negation of R. It is also an extensional predicate, whose definition in the EDB is the complement of R w.r.t. $\mathbb{B}_P$.

2.3.2.2 Logical consequence

We now move up to the general setting, where negation may be applied to any predicate, may it be extensional or intensional. The heart of the operational semantics for standard Datalog previously defined was the T_P operator (see Definition 2.45), which of course needs to account for the negated atoms.

Definition 2.63. Given a program P, the **extended immediate consequence operator** $\tilde{T}_P$ works on interpretations, i.e. $\tilde{T}_P : \mathcal{P}(\mathbb{B}_P) \to \mathcal{P}(\mathbb{B}_P)$. The definition is similar to the previous one, but now only captures extensional predicates in the given (or previous) interpretation ($I|_{EDB}$) and checks that the instances of negated atoms have not been deduced:

$$\begin{aligned}\tilde{T}_P(I) = \{F \in \mathcal{P}(\mathbb{B}_P) \mid F \in I|_{EDB} \ \vee \ F = \mathit{head}(\iota(C)),\ C \in P \\ \wedge\ \mathit{body}^+(\iota(C)) \subseteq I \\ \wedge\ \mathit{body}^-(\iota(C)) \ \cap \ I = \emptyset\}\end{aligned}$$

Remark 2.64. Unlike the standard T_P, this new $\tilde{T}_P$ operator is not inflationary, i.e. it is not the case that $I \subseteq \tilde{T}_P(I)$ for every $I \subseteq \mathbb{B}_P$ (note that I is not restricted to facts about extensional predicates). Still unlike T_P (see Lemma 2.46), it is not monotonic either.

This remark on the monotonicity of $\tilde{T}_P$ is at the heart of the intricacies brought by the negation, as illustrated by the following example.

Example 2.65. We use a program P which only contains a single rule $p \leftarrow \neg q$ and no variable. Several properties of the minimal model semantics (see Section 3.3.1) are violated when trying to apply it to P:

Uniqueness $I_1 = \{p\}$ and $I_2 = \{q\}$ are both minimal Herbrand models, i.e. they are both compatible with the only rule of P, whereas $\emptyset$ is not (the absence of q would imply the presence of p).

Models closed by intersection As explained just above, $I_1 \cap I_2 = \emptyset$, unlike I_1 and I_2, is not a model of P.

Monotonicity $\tilde{T}_P(\emptyset) = \{p\}$ (not decreasing) and $\tilde{T}_P(\{q\}) = \emptyset$ (not increasing)

Beyond these fundamental infringements of the model-theoretic approach we previously relied on, a more immediate and practical issue arises with the use of negation, as shown by the following example.

Example 2.66. Consider the two-rule program $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$. Using the extended

consequence operator of Definition 2.63 and iterating it using Definition 2.48, the results alternate between $\emptyset$ and $\{p, q\}$, indefinitely.

The immediate consequence operator has already been altered to account for the use of negation, but the way it is iterated has not. This is the heart of the solution developed by [Apt et al., 1988], which introduces the notion of **stratified semantics**.

The idea is that the programs are split into strata, defined by the use of negation (the construction of these strata is addressed in Section 2.3.2.3), and that the results of the computation of a stratum are preserved when moving on to the next. This way, the iterations are performed on a *stable* fact base and monotonicty is ensured, which clearly appears in the following definition.

Definition 2.67. Given a complete lattice $\langle \mathcal{L}, \subseteq \rangle$ and a Datalog program P, the **powers of the extended immediate consequence operator** are defined as

$$\begin{cases} \tilde{T}_P \uparrow 0 = \emptyset \\ \tilde{T}_P \uparrow (n+1) = \tilde{T}_P(\tilde{T}_P \uparrow n) \cup \tilde{T}_P \uparrow n \\ \tilde{T}_P \uparrow \omega = \bigcup_{n \geq 0} \tilde{T}_P \uparrow n \end{cases}$$

Example 2.68. Going back to Example 2.66, but using the new iteration of $\tilde{T}_P$, we obtain $\tilde{T}_P \uparrow 0 = \emptyset$ and $\tilde{T}_P \uparrow 1 = \tilde{T}_P \uparrow \omega = \{p, q\}$. In particular, a fixpoint is now reached.

As stated above, for this method to work, the programs need to be stratified.

2.3.2.3 Stratifying a Datalog program with negation

To formally define the notion of stratified program, we need the notion of *predicate definition*, i.e. the set of clauses of a program that define a given (extensional or intensional) predicate.

Definition 2.69. Given a Datalog program P, the **definition of a predicate** p is

$$def(p) \equiv \{C \in P \mid sym(head(C)) = p\}$$

Definition 2.70. Let P be a Datalog program with negation. A **stratification** is a mapping $\sigma : \mathcal{P} \to [1, n]$ that indexes the predicate symbols of P – and can be lifted to atoms – such that, for any clause of the form

$$H \leftarrow B_1, \ldots, B_k, \neg C_1, \ldots, \neg C_m$$

in P, we have

- $\sigma(B_i) \leq \sigma(H)$, for every $i \in [1, n]$
- $\sigma(C_i) < \sigma(H)$, for every $i \in [1, m]$

In other words, a stratification indexes the positive and negative atoms of any clause such that they are bounded and strictly bounded, respectively, by the index of the predicate symbol at the clause's head. This way, we obtain an order in which the different *layers*, or strata, of the program can be computed.

Definition 2.71. Given a program P and a stratification σ, we call P_i a **stratum**, where

$$P_i = \{p \in \mathcal{P} \mid \ \sigma(p) = i\} \text{ and } P_i \neq \emptyset$$

A program P is then partitioned into $P_1 \sqcup \cdots \sqcup P_n$, where $\sqcup$ is the disjoint set union.

Remark 2.72. Given a clause C that appears in the P_i stratum, lifting Definition 2.70, we obtain that any predicate symbol which appears positively (resp. negatively) in the body of C is fully defined by strata which are below or at the same level as (resp. strictly below) P_i. Concretely, for any predicate symbol $p \in \mathcal{P}$,

- if $p \in \bigcup_{L \in body^+(C)} sim(L)$, then $def(p) \subseteq \bigcup_{j \leq i} P_j$
- if $p \in \bigcup_{L \in body^-(C)} sim(L)$, then $def(p) \subseteq \bigcup_{j < i} P_j$

Notation 2.73. Such a stratification of P into $\{P_1, \ldots, P_n\}$ is denoted as $\bar{P}_n$. Each stratum P_i may also be refered to as a program slice.

Remark 2.74. Given a stratification $\bar{P}_n$, every program slice P_i is a semipositive Datalog program w.r.t. the *lower* strata. In other words, assuming we have *computed* slices P_1 to P_{i-1}, and seeing the result as a new EDB, P_i can be dealt with using the method of Section 2.3.2.1.

Remark 2.75. Some Datalog programs, such as the one in Example 2.66, are not stratifiable. We do not go into details in this document, as the technicities of the actual stratification of Datalog programs does not come into play in our work, its justification or explanation. The curious reader will find a synthesis and illustration in Section 6.2.3 of [Dumbrava, 2016], and a more complete version in [Ullman, 1990].

We now have all the tools and notions required to formally define a first semantics of Datalog programs using negation.

2.3.2.4 Iterated Fixpoint Models

As previously intuited (see Remark 2.74), the stratification of a program splits it into a series of components, which will be executed in a sequential manner and *on top* of one another, i.e. using the semantics of the previous slice(s) as the initial interpretaion.

Definition 2.76. Let P be a Datalog program with negation, stratified as $P_1 \sqcup \cdots \sqcup P_n$. The model of P is defined iteratively, using the following relation:

$$\begin{cases} M_1 = \tilde{T}_{P_1} \uparrow \omega \ (\emptyset) \\ M_i = \tilde{T}_{P_i} \uparrow \omega \ (M_{i-1}) \end{cases}$$

Is that setting, the intended semantics of P is M_n.

Example 2.77. We reuse the program of Example 2.55, which defined graph disconnectedness. In that context, the *edge* and *path* predicate symbols can be the first stratum, whereas *disjoint* should go to the second one. The computation of the first stratum returns the set $M_1 = \{edge(1,2), edge(2,1), edge(2,3), path(1,2), path(2,1), path(1,1), path(2,3), path(1,3)\}$

Using this set as the EDB for the second stratum, the semipositive method returns the set $M_1 \cup \{disjoint(2,2), disjoint(3,1), disjoint(3,2), disjoint(3,3)\} = M_2$.

It might be surprising to compute $disjoint(2,2)$ and $disjoint(3,3)$. This is because $path(X,Y)$ is defined as the existence of a *non-empty* path between vertices X and Y. One might be tempted to add a reflexivity using a trick similar to what is shown in Example 2.21.

Remark 2.78. A program may admit more than one stratification (e.g. by using multiple strata for a set of compatible predicates). It is however shown in [Apt et al., 1988] that the semantics of a program is independent from its stratification.

As stated and proved in Chapter 15.2 of [Abiteboul et al., 1995], the stratified semantics is adequate w.r.t. the previously introduced interpretation of Datalog:

Theorem 2.79. For each stratifiable Datalog program P and instance I over $edb(P)$:

- The stratified semantics of P w.r.t. I is a minimal model of P^*, the first-order logic translation of P (see Section 2.2.1), and its restriction to $edb(P)$ equals I
- The stratified semantics of P w.r.t. I is a minimal fixpoint of T_P, and its restriction to $edb(P)$ equals I

Proof. See Section 9.6.1 of [Dumbrava, 2016], also implemented in Coq in [Benzaken et al., 2017b]. □

2.3.3 Alternative semantics

The stratified semantics is used as a reference point in [Dumbrava, 2016], but alternative semantics are introduced and quickly discussed.

Perfect model semantics The perfect model semantics [Przymusinski, 1988] generalizes the notion of stratification introduced above, as it is defined on the level of atoms rather than predicates. The computation of the semantics of a *locally-stratified* program is then similar to the process seen just above.

Stable model semantics The stable model semantics [Gelfond and Lifschitz, 1988], rather than dealing with the negation at the level of the iterated logical consequence operator, transforms the Datalog program. After it has been grounded, rules containing negated facts that appear in the original interpretation I are deleted, as their body can not be satisfied. On the other hand, negated facts which do not appear in I are also deleted, as they are considered *satisfied*.

This leads to a negation-free program, whose semantics can be computed using the standard tools. If this unique minimal model of the transformed program is the original interpretation I, then it is also the unique minimal model of the initial program, called a *stable model*. Given a stratifiable program, this model is the same as the iterated fixpoint model.

Well-founded model semantics The well-founded model semantics [Van Gelder et al., 1991] uses a logical setting that introduces a third, intermediate truth value [Przymusinski, 1990]. The interpretation is then split into two components, founded and unfounded facts, and the logical consequence operator combines the usual immediate consequence with the negation of elements in the greatest unfounded set. For locally stratified programs, the (iterated) well-founded and perfect model semantics are equal, whereas for Datalog programs with unstratified negation, the total well-founded model is the same at the unique stable one.

The takeaway of this section on Datalog extended with negation is that, in practice, reasoning about the negation of a predicate requires a *saturation* of its definition. More concretely, this means that the use of negation may introduce bottlenecks in the execution of Datalog programs, as we will see in Section 6.2.

2.4 Adding on-the-fly constraints

As defined above, Datalog *passes around* values rather than buidling (using complex terms) or introducing new ones, which ensures its finiteness. However, this setting is also a strong limitations for the implementation of many real-life scenarios and problems. Another practical extension of Datalog, which is *lighter* than negation but also comes into play in our work, is the use of non-strictly symbolic predicates.

Definition 2.80. A **primitive predicate** is a predicate which is not defined by rules or the EDB (meaning that it is neither intensional nor extensional), but by actual computations within the Datalog engine.

Example 2.81. Assume we have an EDB filled with people represented as facts of the form $p(N, H)$, where N is the person's name (or id) and H his or her height, and we want to use Datalog to compute the set of heightest people in this EDB. This can be done within the strict frame of Datalog, for example with the rules of Figure 2.6, with a new extensional predicate $greater$. This approach then requires the addition to the EDB of every fact $greater(x, y)$ where x is indeed greater than y and x and y appear as the second argument of a fact about p in the EDB.

$$taller(N_1, N_2) \leftarrow p(N_1, H_1),\ \ p(N_2, H_2),\ \ greater(H_1, H_2).$$
$$tallest(N_1) \quad \leftarrow \neg taller(N_2, N_1).$$

Figure 2.6: Computing the tallest people in *strict* Datalog

This method relies on the finite nature of Datalog programs, and in particular of the set of relevant constants. However, the domain of such a program is usually an *almost infinite set*, in the sense that they can not be naively enumerated in practice – let alone in a quadratic way, as in Figure 2.6. Purely symbolic Datalog, as introduced in this chapter, are then not fit to handle some use cases.

In contrast, individual Datalog engines may support primitive predicates, e.g. efficient implementations of $=$ or $\geq$. Such an engine could run the program of Figure 2.7, which does not require the introduction of new predicates or facts.

$$taller(N_1, N_2) \leftarrow p(N_1, H_1),\ \ p(N_2, H_2),\ \ H_1 \geq H_2.$$
$$tallest(N_1) \quad \leftarrow \neg taller(N_2, N_1).$$

Figure 2.7: Computing the tallest people in Datalog with primitive predicates

Remark 2.82. The addition of primitive predicates should not be at the cost of breaking the finite nature of the execution of Datalog programs. To avoid this caveat, the safety constraint defined in Section 2.1.3, does not take into account the occurrences of arguments in primitive predicates. This way, they can be seen as additional, very convenient constraints added at the level of the engine rather than the actual language.

Chapter 3

Datalog in Coq

> Et maintenant, je me demande : quand vient la nuit, est-ce que la Machine pleure, elle aussi ? Est-ce qu'elle hurle dans un oreiller, comme moi, depuis le fond de sa solitude ?
>
> Emmanuel Denise, Canard PC 396

In her thesis [Dumbrava, 2016], Stefania Dumbrava developed a formalization of Datalog within the Coq proof assistant, called DatalogCert [Benzaken et al., 2017b]. It contains two engines, one for standard Datalog, implementing and certifying the T_P operator as presented in Section 2.2.2, and one for Datalog extended with negation, using the semantics introduced in Section 2.3.2.

Remark 3.1. This chapter introduces the version of DatalogCert corresponding to the paper [Benzaken et al., 2017a] and found at [Benzaken et al., 2017b]. Some of its authors have since developed more complete or alternative versions (see for example [Bonifati et al., 2018]). The work presented in this thesis should be adapted to such newer versions, which should serve as the basis of potential future works.

Our work is also formalized in Coq, using the development of the positive Datalog engine. Although we formally defined the computation of Datalog programs using negation above, it was only to give an intuition of the way Datalog engines work in practice, and how it can raise performance issues, as explained in Section 6.2.

As these Datalog engines are heavily based on the Mathematical Component (MathComp) library, Section 3.1 first introduces the relevant basics with a simple, user-oriented overview. We then present in Sections 3.2 and 3.3 the core syntactic and semantic components of the positive Datalog engine found in DatalogCert, to provide context and tools for our own work.

3.1 Finite types and notations in MathComp

MathComp introduces a type hierarchy for algebra, where refined structures inherit properties and structural functions of their *ancestors* [Garillot et al., 2009]. Figure 1 of [Sakaguchi, 2020] shows the hierarchy of structures found in version 1.10.0 of MathComp. As an illustration, an algebra enthusiast may notice a ring type (`ringType`), which can be refined into the

types of commutative rings (`comRingType`), or rings whose units have computable inverses (`unitRingType`). These two types can then be specialized into the type of commutative rings whose units have computable inverses, `comUnitRingType`.

However, both [Benzaken et al., 2017b] and the work we built upon it do not venture far into the algebraic types introduced in MathComp. They rather use – and, in our case, extend – the generic types which form the backbone of MathComp. We then first need to introduce these types and the possibilities they unlock.

3.1.1 eqType

The basis of this *backbone* is a type called `eqType`. It consists of another type packaged with a decidable equality, denoted as `==`. More concretely, given an `eqType` and two elements of this type, their equality can be computed as a boolean, as shown in Figure 3.1

```
Variable A : eqType.
Variable x y : A.

Check x == y.
```

```
x == y : bool
```

Figure 3.1: Using the decidable equality of an eqType

Conversely, to build an `eqType`, the base method is to provide a boolean equality relation `e`, and show that it enforces the axiom of Figure 3.2, i.e. that its behavior follows the propositional equality.

```
Definition axiom T (e : rel T) := forall x y, reflect (x = y) (e x y).
```

Figure 3.2: eqType axiom

Remark 3.2. To the best of our understanding, the general methodology of MathComp is to fit structures into constrained types such as `eqType`, or the following subtypes, which *unlocks* many definitions and notations. In particular, DatalogCert relies heavily on this approach by leveraging the decidable and finite nature of Datalog.

3.1.2 choiceType and countType

The next two types are `choiceType` and `countType`. The first is the interface of types with a choice operator, i.e. a function that takes a predicate and a witness of its non-emptiness, and returns a standard element of the type satisfying the predicate.

The second type, `countType`, is an interface for countable types, i.e. types which are indexed. More concretely, a `countType` is packaged with an injective function that associates a `nat` index to any element of the type, as well as its partial (if the type contains finitely many elements) inverse.

Remark 3.3. All the types presented in this section form a strict hierarchy, in the sense that a `countType` is a `choiceType`, which in turn is an `eqType`. The most precise and interesting type, which inherits the properties of all the others, is `finType`.

3.1.3 finType

The final interface, `finType`, describes types with finitely many elements. This setting unlocks new possibilities, as having only a finite number of elements in a type allows for terminating iterations. We introduce some Examples of `finType`s and associated notations which will be used in the rest of this document.

Example 3.4. Given an integer `n`, one can define the `finType` of `nat`s strictly lower than `n`, called `Ordinal n` and written `'I_n`. To build a `'I_n` from an integer `x`, one needs a proof that `x < n`. Conversely, such a proof can be extracted from an ordinal.

Example 3.5. Functions with a finite domain, i.e. from a `finType A` to a (generic) type `B` form a type called `finfun`, and written `{ffun A -> B}`. If `B` is also a `finType`, then `{ffun A -> B}` is a `finType` itself.

Example 3.6. Given a type `A` and an integer `n`, `n.-tuple A` is the type of lists over `A` with exactly `n` elements. If `A` is a `finType`, so is `n.-tuple A`.

Example 3.7. Given a finite type `A`, MathComp provides the `finType` of sets over `A`, written `{set A}` (which, by transitivity, allows the definition of `{set {set A}}`, and so on).

Definition 3.8. Such types, along with related operations are axioms, are packed within dependent records usually called *mixins* (see Section 2.1 of [Garillot et al., 2009]). Chapter 7 will show how we define and fit some types into such structures.

3.1.4 Using MathComp types

One of the benefits of MathComp is the introduction of some "paper-like" notations for lists and sets. These notations, which rely on straightforward definitions, allow for much more readable and higher-level definitions and proofs, and are used in Parts IV and V of this thesis.

Remark 3.9. A list is called a "seq" in MathComp. Both terms will be used interchangeably in this thesis.

```
Variable A : eqType.
Variable B : Type.

Variable y : A.
Variable f : A -> B.
Variable s : seq A.
Check [seq f x | x <- s & x != y].
```

```
[seq f x | x <- s & x != y] : seq B
```

Figure 3.3: Browsing, filtering and transforming a list with MathComp notations

Example 3.10. Figure 3.3 shows a notation on lists which mixes `filter` and `map`. All occurrences of `y` are filtered out, using the decidable (in)equality packed within the `eqType` (`filter` expects a `bool` predicate), and a function `f` is applied.

Definition 3.11. The `seq` module in MathComp contains many useful notations and functions. In particular, `x \in s` is the boolean membership of `x` in sequence `s`, and the `all` and `has` functions check that all or at least one element of a `seq` enforce(s) a given predicate. The associated lemmas of these functions, `allP` and `hasP`, are used throughout the rest of this thesis.

```
Variable A : eqType.
Variable s : seq A.
Variable P : pred P.
Variable x : A.

Check x \in s.

Check all P s.
Check has P s.


Check (@allP A P s).
Check (@hasP A P s).
```

```
x \in s : bool

all P s : bool
has P s : bool

            (* forall x, x \in s -> P x *)
allP : reflect {in s, forall x : A, P x} (all P s)
hasP :
  reflect (exists2 x : A, x \in s & P x) (has P s)
```

Figure 3.4: Enforcing predicates in sequences

Example 3.12. The set interface includes many definitions and notations, such as

- set union, intersection, difference and complement, respectively `:|:`, `:&:`, `:\:` and `~:`.
- "big operators" notations, such as `\bigcup_{x in X} f x` (for $\bigcup_{x \in X} f(x)$)
- comprehension notations, such as `[set f x | x in X & p x]`
- the notions of subset and partitions, with the associated lemmas
- decidable quantifications, such as `[forall x in X, P x]` and `[exists x in X, P x]` where `X` is a `{set A}` and `P` a predicate over `A`, which are defined as booleans

Now that we have reviewed the relevant subset of MathComp, we can introduce the positive Datalog engine of [Benzaken et al., 2017b, Dumbrava, 2016].

3.2 Datalog syntax

We proceed as in Section 1.1, i.e. we first present the construction of Datalog programs, and then their manipulation with groundings and substitutions.

3.2.1 Building blocks

Just like the paper syntax, the formalization first assumes sets, seen as `finTypes`, for constants and predicate symbols, as well as an arity function.

```
Variable constype : finType.
Variable symtype  : finType.
Variable arity    : {ffun symtype -> nat}.
```

Figure 3.5: Constants, predicate symbols and arity

As for the variables, they are encoded using ordinals. To do so, the formalization assumes a number of variables:

```
(* the type of variables will be 'I_n *)
Variable n : nat.
```

Figure 3.6: Defining variables in DatalogCert

These parameters are used to build atoms. The formalization separates ground and *normal* atoms at the type level. In both cases, they define a *raw* type, and on top of it the actual, dependent atom type. This type enforces the well-formedness condition, i.e. that the number of arguments in the atom matches the arity of the involved predicate.

```
(* ground atoms *)                  (* predicate / arguments *)
Inductive raw_gatom := RawGAtom of symtype & seq constant.

Definition wf_gatom ga := size (arg_gatom ga) == arity (sym_gatom ga).
Structure gatom : Type := GAtom {uga :> raw_gatom; _ : wf_gatom uga}.

(* generic atoms *)
Inductive raw_atom  := RawAtom  of symtype & seq term.

Definition wf_atom a := size (arg_atom a) == arity (sym_atom a).
Structure atom : Type := Atom {ua :> raw_atom; _ : wf_atom ua}.
```

Figure 3.7: Defining (ground) atoms

With the atoms, clauses and programs can now be defined as well. Note that *normal* clauses and ground clauses are again separated at the type level. Also note that a program is defined as a sequence (MathComp's nomenclature for list) of clauses rather than a set, as in Definition 2.8, because atoms, and thus clauses, are not defined as a `finTtype`. This point is discussed in Section 7.3.

```
Inductive clause  : Type := Clause  of atom  & seq atom.
Inductive gclause : Type := GClause of gatom & seq gatom.

Definition program := seq clause.
```

Figure 3.8: Lifting to full Datalog programs in DatalogCert

DatalogCert introduces straightforward functions, seen in Figure 3.9, to collect variables in terms, atoms and clauses.

```
Definition term_vars t : {set 'I_n} :=
  if t is Var v then [set v] else set0.

Definition raw_atom_vars (ra : raw_atom) : {set 'I_n} :=
  \bigcup_(t <- arg_atom ra) term_vars t.

Definition atom_vars (a : atom) : {set 'I_n} :=
  raw_atom_vars a.

Definition tail_vars tl : {set 'I_n} :=
  \bigcup_(t <- tl) atom_vars t.

Definition cl_vars (cl : clause) : {set 'I_n} :=
  tail_vars (body_cl cl).
```

Figure 3.9: Collecting variables in DatalogCert

These functions are for example used to implement of the safety condition described in Section 2.1.3, as shown in Figure 3.10.

```
(* clause safety: all head variables should appear among the body variables *)
Definition safe_cl cl :=
  atom_vars (head_cl cl) \subset tail_vars (body_cl cl).

(* program safety: all clauses should be safe *)
Definition prog_safe p := all safe_cl p.
```

Figure 3.10: Datalog safety in DatalogCert

3.2.2 Manipulating formulae

A grounding is defined as a `finfun` (see Example 3.5) from variables to constants. Since the codomain is defined as a `finType`, groundings are finite themselves.

```
Definition gr := {ffun 'I_n -> constant}.
```

Figure 3.11: Groundings as finite functions

Like the rest of the syntax, groundings are built inductively, starting with their application to terms. If the given term is already a constant, then it is left unchanged. Otherwise, i.e. if it is a variable, the associated constant w.r.t. the grounding is returned.

```
Definition gr_term (g : gr) (t : term) :=
  match t with
  | Var v => g v
  | Val c => c end.
```

Figure 3.12: Defining term groundings in DatalogCert

The next steps of the definition are straightforward but illustrate the need to work with the well-formedness proofs carried by the atoms, i.e. show that applying the grounding to an atom does not break the property on the number of arguments.

```
(* raw atom grounding *)
Definition gr_raw_atom g ra :=
  RawGAtom (sym_atom ra) [seq gr_term g x | x <- arg_atom ra].

(* lift to full atoms *)
Definition gr_atom_proof g a : wf_gatom (gr_raw_atom g a).

(* Building an atom with a well-formedness proof *)
Definition gr_atom g a := GAtom (gr_atom_proof g a).

(* clause grounding *)
Definition gr_cl g cl :=
  GClause (gr_atom g (head_cl cl)) [seq gr_atom g a | a <- body_cl cl].
```

Figure 3.13: Lifting groundings to atoms and clauses

```
Definition sub := {ffun 'I_n -> option constant}.

Definition sterm s t : term :=
  match t with
  | Val d => Val d
  | Var v => if s v is Some d
               then Val d
             else Var v end.

Definition sraw_atom ra s :=
   RawAtom (sym_atom ra) [seq sterm s x | x <- arg_atom ra].

Lemma satom_proof a s : wf_atom (sraw_atom a s).

Definition satom a : sub -> atom := fun s => Atom (satom_proof a s).
```

Figure 3.14: Defining substitutions in DatalogCert

The substitutions work similarly, the difference being that a variable is not necessarily mapped to a constant, i.e. the codomain is an option type and the term substitution may leave a variable untouched if it is mapped to `None`. The concrete changes in the application of substitutions are found at the level of terms, i.e. function `sterm`.

Another change is the fact that substitutions can be compared in a more fine-grained manner than groundings. Figure 3.15 introduces the `s1 \sub s2` notation, which checks that substitution `s2` extends `s1`, i.e. contains at least the same mappings to constants.

```
Definition sub_st s1 s2 :=
  [forall v : 'I_n, if s1 v is Some b1 then (v, b1) \in s2 else true].

Notation "A \sub B" := (sub_st A B).
```

Figure 3.15: Comparing substitutions in DatalogCert

The functions of Figure 3.14, due to the potentially incomplete nature of substitutions and unlike `gr_atom`, return atoms rather than ground atoms. Datalogcert also provides an application of a substitution, shown in Figure 3.16, that expects a constant `def` to *fill the blanks*, and returns a ground atom.

```
Definition gr_term_def s t : constant :=
  match t with
  | Val c => c
    (* odflt d x returns x if it is of the form Some y, d otherwise *)
  | Var i => odflt def (s i)
  end.

Definition gr_raw_atom_def s ra : raw_gatom :=
  RawGAtom (sym_atom ra) (map (gr_term_def s) (arg_atom ra)).

Lemma gr_atom_def_proof s a : wf_gatom (gr_raw_atom_def s a).

Definition gr_atom_def s a : gatom := GAtom (gr_atom_def_proof s a).
```

Figure 3.16: Grounding with a substitution

3.3 Semantics

Like the paper definition of Datalog (see Section 2.2), DatalogCert contains and implements both the Minimal Model and Fixpoint semantics. The former, more abstract, is used as a reference in the certification of the latter, more applicative one, which is ultimately extracted and exported as the actual Datalog engine.

3.3.1 Minimal Model Semantics

With the formalization of substitutions, the minimal model semantics as defined in Section 2.2.1 is implemented in a concise and clear manner, shown in Figure 3.17. The authors first

define the satisfaction of a clause w.r.t. a Herbrand Interpretation (denoted as `interp`), then the notion of Herbrand model at the level of clauses and programs.

The implementation of minimal model semantics is itself rather minimal, which reflects its fundamental aspect. In comparison, the translation of the fixpoint semantics requires more work.

```
(* Head and body of a ground clause *)
Definition head_gcl gcl := let: GClause h b := gcl in h.
Definition body_gcl gcl := let: GClause h b := gcl in b.

(* An interpretation is a set of ground atoms *)
Notation interp := {set gatom}.

(* If every ground atom in the body is in the interpretation,
   then so is the head *)
Definition gcl_true gcl (i : interp) : bool :=
  all (mem i) (body_gcl gcl) ==> (head_gcl gcl \in i).

Definition cl_true cl i := forall g : gr, gcl_true (gr_cl g cl) i.

Definition prog_true p i :=
  forall g : gr, all (fun cl => gcl_true (gr_cl g cl) i) p.
```

Figure 3.17: Minimal model semantics in DatalogCert

3.3.2 Fixpoint Semantics

The implementation of the Fixpoint Semantics can be split into two components. In Definition 2.45, the T_P operator tries out every possible substitution to build new facts. This method would be easy to define in Coq using MathComp's set notations and the finiteness of the substitution or ground types, but the efficiency of the extracted Datalog engine would then be seriously impacted.

In consequence, rather than actually matching any substitution, the engine builds the minimal set of substitutions that, by construction, match the given clause and interpretation. Since this *constructive matching* will be used and discussed in our own work, we first introduce all the relevant definitions.

For shortness and clarity, the completeness and soundness results are mentioned but we do not show or discuss their formalization and proof (you may find these informations in Section 8.5.1 of [Dumbrava, 2016]). We then quickly outline how the certification of the fixpoint semantics, i.e. its relation to the minimal model one, is stated.

3.3.2.1 Constructive matching of clause bodies

The constructive matching is developped in a bottom-up fashion. The first step, at the level of terms, has three arguments: a term `t`, an *expected constant* `d` against which `t` is matched, and a substitution `s` that will (potentially) be enriched to store the result of the match.

If `t` is the same as `d`, then no addition to `s` is required, and the substitution is returned. On

the other hand, if `t` is a constant but not the same as `d`, the match fails and `None` is returned. If `t` is a variable `v` that is already mapped to a constant by `s`, a similar equality check occurs. Finally, if `s` does no associate `v` to a constant, the substitution is enriched with its mapping to the *expected constant* `d` and returned.

```
(* matching a term [t] against a constant [d],
   starting from an initial substitution [s].  *)
Definition match_term d t s : option sub :=
  match t with
  | Val e => if d == e then Some s else None
  | Var v => if s v is Some e then
             (if d == e then Some s else None)
             else Some (add s v d)
  end.
```

Figure 3.18: Constructive term matching in DatalogCert

This role of the *expected constant* appears more clearly in the next function, which defines the matching between an atom and a gatom. The two lists of arguments (terms for the atom and constant for the gatom) are *zipped*, meaning that they are browsed in parallel. At each step, the constant from the gatom is used as the *expected constant* for the `match_term` function. The substitution is enriched step by step, using a fold. Although it is called `match`, this operation may then be better understood as a unification procedure between an atom and a ground atom.

```
Definition match_raw_atom s ra rga : option sub :=
  match ra, rga with
    | RawAtom s1 arg1, RawGAtom s2 arg2 =>
      if (s1 == s2) && (size arg1 == size arg2)
      then foldl (fun acc p => obind (match_term p.1 p.2) acc)
                 (Some s) (zip arg2 arg1)
      else None
  end.

Definition match_atom s a (ga : gatom) := match_raw_atom s a ga.
```

Figure 3.19: Constructive atom matching in DatalogCert

The matching between an atom and a ground atom can be lifted to atom and interpretation, i.e. set of ground atoms. To do so, `match_atom` is called on each pair, and the substitutions which were successfuly computed are collected.

```
Definition match_atom_all (i : interp) a s :=
  [set x | Some x \in [set match_atom s a ga | ga in i]].
```

Figure 3.20: Constructive matching of a set of atoms in DatalogCert

This way, `match_atom_all i a s` returns the set of substitutions built upon `s` such that, when applied to `a`, produce a ground atom in `i`. The next step is to define the matching of

a full clause's tail, still w.r.t. an interpretation.

To do so, a join and a monadic fold for the set monad are defined. Concretely, function `bindS` applies `f : A -> {set B}` to each element of a set `i` and flattens the result (`cover` is the union amongst a set of sets). The monadic fold applies `f` to all elements of the `l` list, using the `s0` value for its first iteration.

```
Definition bindS {A B : finType} (i : {set A}) (f : A -> {set B}) : {set B} :=
  cover [set f x | x in i].

Fixpoint foldS {A : Type} {B : finType}
         (f : A -> B -> {set B}) (s0 : {set B}) (l : seq A) :=
  if l is [:: x & l] then bindS s0 (fun y => foldS f (f x y) l)
  else s0.
```

Figure 3.21: Folding and flattening sets

This special fold is applied to `match_atom_all`, starting with a set only containing the empty substitution. This way, at each new atom, every substitution computed so far is (potentially) expanded into a new set of enriched substitutions, and the result is flattened.

```
Definition match_body i (tl : seq atom) : {set sub} :=
  foldS (match_atom_all i) [set emptysub] tl.
```

Figure 3.22: Constructive tail matching in DatalogCert

The `match_body` function returns the set of substitutions which, when applied to the provided tail (i.e. list of atoms), results in a list of ground atoms that all appear in the considered interpretation. This function is also generalized by `match_pbody`, which expects an initial set of substitutions rather than using a singleton with the empty substitution. Section 9.3 discusses how we tried to use it for inductions which could not be performed on `match_body`.

```
Definition match_pbody tl i ss0 := foldS (match_atom_all i) ss0 tl.
```

Figure 3.23: Generalization of tail matching

The constructive matching being fully defined, it can be used in the definition of the actual T_P operator.

3.3.2.2 Certified T_P operator in Coq

DatalogCert splits T_P into two functions. The first, `cons_clause`, computes the deductive part of the operator.

```
Definition cons_clause (def : constant) (cl : clause) i : {set gatom} :=
  [set gr_atom_def def s (head_cl cl) | s in match_body i (body_cl cl)].
```

Figure 3.24: Deduction of new facts with a clause

Remark 3.13. Note that the matching of the computed substitutions is not checked. As previously stated, `match_body` builds a set of substitutions that, by construction, match clause `cl` and interpretation `i`.

Remark 3.14. The `def` argument is here simply for typing purposes, i.e. to compute a set of ground atoms rather than normal atoms (see the end of Section 1.1.2). Its value is irrelevant, as `match_body cl i` produces substitutions that actually associate a value to each variable appearing in clause `cl`, meaning that `def` will never be used in practice.

The `cons_clause` function is then encapsulated into `fwd_chain`, which both enforces the preservation of the interpretation and applies the deduction to every clause in the program.

```
Definition fwd_chain def p i : {set gatom} :=
  i :|: \bigcup_(cl <- p) cons_clause def cl i.
```

Figure 3.25: Iterating deduction over a program

Although we do not recall the lemmas certifying every component of Datalogcert's implementation of the fixpoint semantics, we do state the most general result, which relates both semantics, in Figure 3.26.

```
Lemma incr_fwd_chain_complete (s0 : {set gatom}) :
{ m : {set gatom} &
{ n : nat | [/\ prog_true p m
              , n = #|bp|
              , m = iter n (fwd_chain def p) s0
              & forall (m' : {set gatom}), s0 \subset m'
                -> prog_true p m' -> m \subset m']}}.
```

Figure 3.26: Adequacy of the fixpoint semantics

This lemma states that iterating $T_P\,\mathbb{B}_P$ times, i.e. as many times as there are possible atoms, captures the whole semantics `m` of the program while being minimal.

Part III
Network Verification

The introduction of this thesis discusses the increasing need for safety and security in the design and maintenance of networks, and the resulting introduction of formal methods in the field of networking.

Chapter 4 unboxes the concept of network verification and outlines existing tools. Then, Chapter 5 focuses on a Datalog-based tool called Network Optimized Datalog (NoD). Finally, Chapter 6 introduces a tool built on top of NoD, and explains how its development led us to work on optimizations for the Datalog language.

Overall, this aim of this part is simply to familiarize the reader with the notion(s) of network verification, the various approaches from formal methods that have been leveraged so far, and the way they challenge traditional Datalog engines. The many intricacies and details of networking are then left out for shortness and clarity, and their knowledge should not be required.

Chapter 4

Approaches to network verification

> Et la causerie, descendant des théories élevées sur la tendresse, entra dans le jardin fleuri des polissonneries distinguées
>
> Guy de Maupassant, *Bel-Ami*

Historically, the design and implementation of networks has not relied on the use of formal methods. As an illustration, the authors of [Doenges et al., 2021] recall the words of internet-pioneer David D. Clark: "we believe in rough consensus and running code", which they analyze as a reflection of the notion of robustness upon which networking is built. Basically, robustness is not seen as a formal specification coupled with a proof that the analyzed system does not deviate from it, but rather a system which may contain small deviations from its general design, as long as they do not threaten the intended behavior.

As another illustration of the way priorities are established within the networking community, the seminal paper [Clark, 1988] states "While tools to verify logical correctness are useful, [...] they do not help with the severe problems that often arise related to performance".

The bottom-up philosophy described just above has been challenged over the last ten to fifteen years, with the introduction of Software-Defined Networking, or SDN [Hu et al., 2014]. This framework separates the forwarding plane, i.e. the switches and their forwarding rules, from the control plane, i.e. the higher-level routers which establish these local rules to implement a network-wide policy. As the name indicates, this approach roughly aims at designing networks like software, i.e. with programming languages rather than at the hardware level.

As a side note, in his talk at PEPM'20[1], Nate Foster presents SDN as a form of partial evaluation, in the sense that the network-wide program that eventually computes the actual configuration of all devices is specialized for the underlying topology. This notion is not completely disjoint from the ideas we design and implement in Part V.

This top-down approach of networking opens the way to more traditional, higher-level verification tools, and even to the synthesis of network configurations. Section 4.1 first outlines the specificities of networking with which researchers struggle. Then, Sections 4.2, 4.3 and 4.4 introduce the three main types of network verification problems, i.e. dataplane verification, control plane verification and network synthesis, as well as some tools and techniques they harness.

[1] https://www.youtube.com/watch?v=dHLa1SMILik

4.1 The difficulty of network verification

The field of network verification is inherently difficult. One of the most obvious reasons is what networks fundamentally are, i.e. highly distributed systems, the complexivity of which is well-known. However, there are also some specificities in the way networks are designed and built, which we quickly outline for context.

4.1.1 A distributed and opaque development

The internet, like numerous other systems, relies on many protocols. These protocols, such as BGP, GRE or MPLS, are defined in documents called *Request For Comments* [rfc, 1989, Li et al., 2000, Viswanathan et al., 2001], or RFCs, which are published by the *Internet Engineering Task Force*. At the time of writing, there are over 8,500 RFCs, which are all informal [Doenges et al., 2021].

On the other hand, some documentations try to be exhaustive, but lack the theoretical tools (or habit) to provide an efficient formalization of both the actual systems and their semantics. Moreover, many such systems are developped iteratively, leading to an inflation in the sheer size of the documentation. These two caveats are illustrated by the documentation of the OpenFlow protocol [McKeown et al., 2008], which grows from 44 pages[2] to 165[3], and then to a rather obstruse set of 283 pages[4], where the protocol is fundamentally presented in the form of C code – which makes it *de facto* the language used to describe its semantics.

The points discussed so far in this Section are very elegantly summarised in [Shenker et al.,], which first recalls that *mastering complexity* and *extracting simplicity* are two very different tasks, which do not rely on the same abilities. It then states that networking never made the distinction, and historically focused strictly on mastering complexity. Overall, it advocates for the development of the intellectual foundations and capacity of abstractions required to shift from complexity to simplicity.

This absence of solid and clear foundations led to the existence of many protocols, which are hard to study and justify in the absence of formalization, and whose implementation is subject to interpretation. These implementations then vary across vendors and internet providers, meaning that verification tools must be flexible in the specifications they check or enforce, but at the same time be as automatic and precise as possible.

4.1.2 Packet-level combinatorics

This matrix of heterogeneous implementations is built on top of another combinatorics issue in networking, which is the size of packet headers.

Each packet contains not only the pure data it is supposed to carry around, but also a header, whose content roughly encodes where the packet is coming from, where it is going, some informations on the actual content and so on (in that sense, at the level of said content, this header is akin to *meta-data*, whereas it could be thought of as a type at the protocol level). This header is used by the various protocols, e.g. some bit is set to 1 when the packet goes through some required network component, to store the information.

[2] https://opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf
[3] https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.3.pdf
[4] https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

In other words, whereas the actual data carried by a packet is of no interest to determine its behavior, the entierity of the header must be taken into account throughout its flow in the analyzed network, rewritings included. An IPv6 packet header has *at least* 40 bytes, i.e. 320 bits, which can be extended by the various protocols involved. This means that, in practice, a naive representation of packet headers will lead to a state explosion of the tool's underlying model. This precise point is crucial for network verification in general, Sections 5.3 and 6.2 discuss it in-depth in the context of the Network-Optimized Datalog and Octant tools.

4.1.3 A variety of questions

So far, we have used the phrase *network verification* without explaining what are the properties one may want to check and enforce. There is a multitude of such questions, among which

- Can a packet flow from A to B?
- Are sub-networks X and Y strictly isolated?
- Are packets efficiently shared across the different possible paths?
- Will my network still be operational if up to n links fail?
- Do all packets from sub-network X go through a network component which performs operation P?
- Do two networks have the same behavior?

The questions in network verification are then very diverse in nature, meaning that a (theoretical or practical) tool may be relevant for some of them, but not all, which complicates matters. As a sidenote, Section 5.2 discusses how the expressivity and flexibility of Datalog allow it to formalize and efficiently answer many of such questions.

With the exception of the last one, the questions above belong to the field of dataplane verification, which the next section discusses. Then, Section 4.3 introduces control plane verification, which includes the last question of the list and can roughly be seen as a kind of *a priori* verification process. Finally, Section 4.4 discusses the synthesis of correct-by-construction networks.

4.2 Dataplane verification and testing

In networking, the dataplane – sometimes also called forwarding plane – is the set of low-level mechanisms that define the way packets are forwarded and transformed. For example, the authors of [Fayaz et al., 2016] see it as a function of the form $(paquet, port) \rightarrow (paquet, port)$.

More concretely, it is the topology and the forwarding tables contained in the switches spread across the network. These forwarding tables (very) roughly contain rules of the form "if a packet has a header that matches pattern t, it should rewrite its i^{th} and j^{th} bits to 0 and go through port p", which are assigned priority levels. The sum of these very local rules, along with other specialized components (firewall, network functions and middleboxes), define the behavior of the full network.

Dataplane verification covers any task that analyzes the properties of such a given configured network. This section outlines some tools that address these questions.

4.2.1 Finding a counterexample

One of the earlier works on network verification is [Xie et al., 2005], which tries to formalize and give a higher-level view of network reachability. After that, part of the research efforts focused on encoding networks into existing verification engines to automatically perform such analyses. Since these engines were not designed to handle networks and their specificities, the encodings had to be carefully chosen to allow scalability – eventually, specific engines were introduced, as discussed in the rest of this Chapter and the next two.

Such an example is found in FlowChecker [Al-Shaer et al., 2009, Al-Shaer and Al-Haj, 2010], a model-checking tool that uses Computation Tree Logic [Clarke et al., 1986] as the specification language, and NuSVM [Cimatti et al., 2000] as its backend engine. Flowchecker has only been used in small experiments and, despite smart encodings such as the use of Binary Decision Diagrams [Bryant, 1986] to perform the matching mechanism within forwarding tables, it does not scale to industrial-size uses.

The Z3 SMT-solver [de Moura and Bjørner, 2008] is another classical verification tool used for network verification. Its use as a backend for a specific Datalog-based verification tool is discussed in-depth in Section 5.3, but it had previously been used more directly in [Zhang et al., , Zhang and Malik, 2013]. In that setting, the network is encoded as a formula N and the negation of the property to enforce is encoded as $\neg P$. Then, the satisfiability of $N \wedge \neg P$ is checked, and Z3 provides a counterexample to P w.r.t. N if it is valid.

This work is optimized by focusing on the behavior of a single packet on a single path, hence the return of a single counterexample. A very similar method is used in [Mai et al., 2011]. On the other hand, some tools use abstraction to handle at once a set of packets to cover all the possible paths.

4.2.2 Finding all counterexamples

Although finding a counterexample to a specification or having a proof that there is none is, in itself, a valuable information, this approach has some limitations. The main one is that a single counterexample is not always very helpful – and even sometimes misleading – to fix a system as complex and intricate as a network configuration.

For example, computing the set of all counterexamples rather than a single one fosters incremental verification. When two versions of a configuration are checked, the singular counterexamples computed may have nothing in common, whereas sets of counterexamples are comparable, e.g. if one is a subset of the other, we may deduce that one of multiple problems has been fixed (or introduced). Network Optimized Datalog [Lopes et al., 2015] belongs to this category, but is discussed in length in Chapter 5.

Moving from a computation of one to all counterexamples changes the underlying problem, which goes from SAT to AllSAT. Modern SAT solvers are not optimized for AllSAT, meaning that smart representation of the domain (mainly packet headers) must be leveraged.

The prime example of this approach is Header Space Analysis [Kazemian et al., 2012]. This tool relies on the observation that reachability verification is more efficient when performed on

equivalence classes rather than individual packets [Yousefi et al., 2020]. To represent a group of packet headers, ternary bit vectors (e.g. $1 \star \star 0$) are used. Networking boxes (switches, firewalls etc) are seen as functions that take a *packet pattern* and an entry port, and return a set of pairs of packet and output port, i.e. $T : (h, p) \rightarrow \{(h_1, p_1), ..., (h_n, p_n)\}$. The behavior of a network is then seen as the composition of (lifted) routing functions, e.g. $T_3(T_2(T_1(h, p)))$.

According to its own authors, this approach worked terribly until they used a better representation of packet headers called Differences of Cubes [Bjørner and Varghese, 2015], which are formally introduced and discussed in Section 5.3. This representation roughly allows an efficient definition of mechanisms of the form "if a packet header matches shape s, but not exceptions e_1, e_2 and e_3, then...".

Other optimizations have been introduced. NetPlumber [Kazemian et al., 2013] computes a dependency graph that relates forwarding rules to allow incremental updates and parallelization, SecGuru [Jayaraman et al., 2014, Bjørner and Jayaraman, 2015] tries to detect locally that different nearby routers will forward clusters of packets the same way rather than relying on costly propagated analyses, VeriFlow [Khurshid et al., 2013] observes and leverages the fact that the number of header equivalence classes is small in practice, and [Plotkin et al., 2016] eliminates redundancy and reasons up to *network symmetry* (quotients parallelizable processes) to work on a simplified model of the analyzed network.

Finally, like traditional program analysis, recent research in dataplane verification dwelve in probabilistic territory. For example, Netter [Zhang et al., 2021] translates a dataplane into a probabilistic network (which itself encodes a finite discrete Markov chain) and harnesses existing model-checking tools such as PRISM [Kwiatkowska et al., 2011].

4.2.3 Testing

Although exhaustive verification is much more powerful, testing can provide a first, easier analysis of the correctness of a program. One of the main measures by which a test set is evaluated is coverage, i.e. the fact that these tests at least go through every line of the analyzed program, although not with every possible configuration.

Symbolic execution is a technique that maximizes coverage. To do so, a program is seen as a decision tree, where the nodes are its conditionals. Then, the constraints across all paths are collected, and solvers generate test values for each set of constraints. The main tool that uses symbolic execution is Klee [Cadar et al., 2008], which has been developed for LLVM. This technique has been used for dataplane testing [Zeng et al., 2012, Dobrescu and Argyraki, 2014], by replacing program lines with rules and test cases by packet headers.

4.3 Control plane verification and testing

In networking, the control plane is the set of routers and protocols that set up the dataplane. Roughly, they explore and learn the (physical) topology and paths, compute the actual forwarding rules that are installed in the different switches to enforce the given configuration, implement dynamic updates and so on. Even more informally, if the data plane is the muscle of the network, the control plane is its brain. In that setting, the goal of control plane verification is ensuring that, given a collection of router configurations, the resulting dataplane will enforce a given property [Beckett et al., 2018].

Formal analysis and verification of networking protocols is actually older than the notion of SDN – as are networks themselves –, but the problems and techniques have significantly changed since, following the evolutions of telecommuncations and formal methods. Let us only jestingly mention two lines of research pursued at CNET Lannion, now Orange Labs Lannion, a few decades ago. The first is a mix of model-checking techniques *à la* Sifakis [Queille and Sifakis, 1982] and the simulation tool Véda [Jard et al., 1987, Monin, 1989], which was developed using Prolog. The second, led by our other PhD advisor, applies Coq to the verification of distributed protocols [Heyd, 1997].

Going back to SDN, first research works focused on pen and paper, non-automated analysis of the highly complex BGP protocol [Rekhter et al., 2006]. For examples, works such as [Chang et al., 2003, Griffin and Wilfong, 1999] study the instability and possibility of loops, [Gao and Rexford, 2001] develops criterias to avoid loops in the computation of forwarding tables, [Le et al., 2008] showed that the route redistribution technique (allowing routes to be imported from one routing process into another process on the same router) may cause loops.

As for the automated tools, they can be split into two categories [Beckett et al., 2018]. The first category is verification not based on formal semantics model, such as checking configurations against a set of good practices and syntactic patterns [Feamster and Balakrishnan, 2005]. In a more modern fashion, [Bauer et al., 2011] uses machine learning to find such dubious configurations.

Regarding semantics-based approaches, one of the main examples is BatFish [Fogel et al., 2015]. This tool uses Datalog to specify and check the dataplane that would be generated by a set of router configurations w.r.t. a given environment or scenario. However, Batfish can only check the control plane for a single context, meaning that checking the robustness of a control plane w.r.t. a realistic set of possible environments and scenarios is not feasable in practice.

Subsequent tools have tried to address a higher-level verification problem, i.e. checking properties about many or all dataplanes that may emerge from a given control plane, although often at the cost of network design coverage. For example, Bagpipe [Weitz et al., 2016] performs a symbolic execution of the message-passing semantics of BGP in all possible environments, but makes strong assumptions about the underlying network.

Another example is ARC [Gember-Jacobson et al., 2016], which abstracts the configurations as weighted graphs, allowing the consideration of many failure scenarios at once. However, this abstraction is only possible if some features of BGP are not used.

Vericon [Ball et al., 2014], in the spirit of (traditional) program verification, analyzes SDN programs *à la* Floyd-Hoare-Dijkstra using (partially manual) deductive reasoning and first-order logic, with Z3 as a back-end. However, it only checks safety properties and requires manual invariants for the proofs. More recently, Minesweeper [Beckett et al., 2017] and CrystalNet [Liu et al., 2017] both introduce various heuristics to avoid performing eager computations, or work on circomscribed spaces and infer the effects of propagations.

Finally, the control plane may also be subject to testing rather than verification. Klee remains a major tool in this context, and is used to uncover latent bugs in BGP configurations before they appear in the dataplane. On the other hand, the dataplane testing tools introduced in Section 4.2.3 can not handle the level of complexity of control plane models. As another example, NICE [Canini et al., 2012] avoids the state-space explosion described in Section 4.1.2 by analyzing the routing code within the control plane to extract practical equivalence classes on the packet headers.

4.4 Synthesis of correct-by-construction networks

The introduction of Section 4.1 quickly mentions the inherently distributed nature of networks as a general difficulty for verification. It is also true at the stage of development, as a high-level, network-wide behavior has to be manually implemented as a collection of interacting local systems (the forwardings tables in the switches) which *communicate* using catch-all stacks (packet headers) rather than ad-hoc, clearer structures.

This hard and error-prone process is reminiscent of very low-level programming, e.g. in an assembly language, where the stack is used to encode in a sometimes obstruse way high-level concepts. The solution to this situation is also similar, as the introduction of Software-Defined Networking (SDN, see the introduction of this Chapter) is, in part, the research community's advocacy for the introduction of abstraction layers [Casado et al., 2007].

Some early, lightweight abstractions are found outside of the setting of SDN, for example in the introduction of templates and vendor-neutral configuration languages, e.g. RPSL [Kessens et al., 1999], Yang [Björklund, 2010] and Netconf [Enns et al., 2011]. These tools provide some notion of consistency and mitigate one of the difficulties of networking, i.e. the diversity of hardware vendors and associated implementations. However, these languages are not fundamentally different from previous configurations languages and do not address the gap between hardware and high-level intents.

On the other hand, the Frenetic project[5] aims at the development of declarative network programming languages that would allow reasonning about the behavior of the network at a suitably high level of abstraction, and even formally establishing the correctness of the associated compiler and run-time system.

The people behind this project first introduced the homonymous language [Foster et al., 2011]. The two main features of the Frenetic language are the introduction of constructs to read the state of the network and specify forwarding policies, and the modularity allowed by the introduction of policy combinators, e.g. parallelization. It is then refined in [Monsanto et al., 2012], which introduces an actual policy language called NetCore, allowing more expressiveness and modularity. The run-time system is also extended and leveraged to handle features that can not be translated efficiently into forwarding tables, such as intricate packet classifications that could only be encoded in a switch using billions of prefix matching rules.

NetCore is then refined again in [Monsanto et al., 2013], which notably introduces sequential composition, as well as a Python implementation of these abstractions called Pyretic. As a sidenote, Pyretic serves as a basis for Kinetic [Monsanto et al., 2013], which leverages its compositional features to express network policies as finite-state machines, which allows the use of previously existing verification methods.

Finally, [Guha et al., 2013] provides a network-wide semantics to NetCore and a Coq proof of the compiler and run-time system. The results of the first real deployment of this language are also presented.

Although definitely a step in the right direction (i.e. importing programming language theory into networking), the Frenetic project has been built in an iterative fashion, meaning that it lacks a clear metatheory and direction. Moreover, although modular and equipped with theoretical foundations, the languages of this collection define the behavior of the switches, meaning that an analysis of the network-wide behavior must be *extracted* from a low-level

[5] http://frenetic-lang.org/

specification.

To build more solid foundations for network synthesis, the same authors introduce NetKat [Anderson et al., 2014], a network programming language that relies on Kleene Algebra with Tests (KAT, see [Kozen and Smith, 1996]). In practice, the behavior of a network is then specified as regular expressions augmented with a packet algebra that encodes packet matching and rewritings.

This well-studied theory comes with some results and algorithms, such as the decidability of equivalence between two programs (seen as automata). Many of the usual network questions (see Section 4.1.3) can then be encoded into equivalences between the analyzed program and specific, minimal programs. For example, isolation between X and Y in network N can be stated as the equivalence between a packet that goes from X to Y (or the other way around) in N, and the empty program. Since then, NetKat has received various optimizations [Foster et al., 2015, Smolka et al., 2015], been enriched with Linear Temporal Logic [Beckett et al., 2016], and gone into probabilistic territory [Foster et al., 2016].

However, over the years, the tool that has gained the most attention is the P4 domain-specific language. P4 combines high-level abstractions (packet parsers, match-action contructs) with an efficient compiler [Bosshart et al., 2014]. However, like many other networking tools, it has been largely developped within the industry without much consideration for theoretical or formal foundations, as illustrated by its current 163-page documentation that leaves many aspects of the semantics unspecified[6].

Since its introduction, there have been multiple enhancements to P4, the most notable being the recent development of fully formal foundations [Doenges et al., 2021]. More precisely, this work presents a full definitional interpreter for the language as well as a simple core calculus with formal syntax, typing and operational semantics. Moreover, the type soundness and termination of the calculus are proved, and an implementation is developed and tested.

This last and, again, very recent work on a highly popular networking programming language allows one to hope that this field will continue to take inspiration from more traditional language theory and harness decades of research to build safer networking tools and foundations.

[6] https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf

Chapter 5

Network Optimized Datalog

> On ne doit rien croire sans preuve dans ce monde, il faut tout pouvoir prouver ! [...] C'est désespérant !
>
> Kohi Kumeta, *Sayonara Monsieur Désespoir* (tome 3), traduit du japonais par Vincent Zouzoulkovsky

This Chapter focuses on Network Optimized Datalog, or NoD, a dataplane verification tool tuned to check reachability properties of dynamic networks expressed *à la* Datalog using the Z3 SMT-solver [de Moura and Bjørner, 2008, Hoder et al., 2011] as a back-end. It models network policies at a higher-level of abstraction than other tools (e.g., VeriFlow [Khurshid et al., 2013]) and handles dynamicity, in the sense that it is resilient to various changes in the modeled network without requiring changes to internals. This Chapter tries to convey the main ideas behind NoD, and is based heavily on [Lopes et al., 2013, Lopes et al., 2015].

We first discuss in Section 5.1 the choice of Datalog as a specification language, and outline in Section 5.2 the modelization of various network policies with Datalog. We finally introduce in Section 5.3 the modifications made to the underlying Datalog engine for NoD to scale, focusing on a specific component that will be of importance in the next Chapter.

5.1 Datalog as a specification language for network behavior

The authors of [Lopes et al., 2015] identify five features that should be provided by an ideal network verification tool. Three of these features are natively found in Datalog, whereas the other two require some more work.

The first identified feature is that, when computing reachability, one wants to find *all* packet headers that can go from a network element A to B. Classical model checkers and SAT solvers [Biere et al., 2009, Jhala and Majumdar, 2009] can go from *existential* to *universal answers* by adding the negation of the provided solution and iterating, but the performances would not be satisfactory. In contrast, once reachability has been encoded into it (see Section 5.2.1), and given a set of starting packets and locations, Datalog will natively compute the set of all reachable configurations (packet header h at port p).

The second desired feature offered by Datalog is the availability of higher-level constructs,

basically boolean operators. More concretely, the use of multiple rules to define a single predicate is akin to disjunction and can be used for example to combine reachability sets ("a packet can access location L_1 from L_2 using path P_1 *or* path P_2"), whereas the bodies of Datalog rules are conjunctive in nature, which notably allows the expression of forwarding rule priorities ("A packet P can follow this switching rule *if* it matches it *and* there is *no* higher priority matching rule") – although not necessarily in the best and clearest fashion, as we will discuss in Section 6.1.

Finally, the authors underline that, using predicate arguments, Datalog can encode a notion of state. This can in turn be used to model evolving networks (zone failures, packet format changes) and protocols. The next section will emphasize these three benefits of Datalog by showing how it is used to specify the semantics of a network in [Lopes et al., 2015].

In contrast, Datalog does not handle natively the last two desired features: the ability to model the **rewriting** of **large** packet headers. Checking the semantics of a network implies reasoning about headers of around 80 bytes, whose rewritings impact the general behavior of the network. Concretely, the gigantic header space requires a compact representation that still allows dynamic rewriting. Section 5.2 illustrates how the rewriting is modeled at the level of Datalog specifications.

Although one can leverage the fact that many of the bits in a header do not matter to determine the way the packets moves through the network, the amount of information to track and account for remains important, where handling as much as 3 or 4 bytes leads to a state explosion. The authors of [Lopes et al., 2015] then introduce modifications to the Datalog engines, which are presented in Section 5.3.

5.2 Datalog modelization of network beliefs

We re-introduce the core example of [Lopes et al., 2015], which shows how Datalog can be used to specify and verify network reachability, the cardinal problem of network verification. We will then overview some other examples of the paper, which illustrate – still from a very high-level point of view – the main ideas behind the modelization of various network policies one may want to enforce and check.

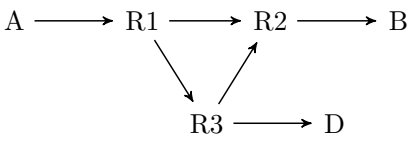
5.2.1 Reachability

We use the topology of Figure 5.1a, where R1, R2 and R3 are routers, whereas A, B and D are end-points. For simplicity, the example assumes that packets carry only two attributes in their headers, *dst* and *src*, which are both encoded over 3-bit vectors.

The industry standard in terms of memory management within routers and switches is Ternary Content-Addressable Memory, or TCAM [Lakshminarayanan et al., 2005]. This hardware-level consensus has driven the design choices in forwarding, notably having the possibility to match only a subset of the bits of a packet header. Concretely, such matchings rules can then contain an "any" bit denoted as $\star$.

Figure 5.1b displays the forwarding table of the example, which contains rules such as described above. Routers and switches may also rewrite some bits, as illustrated by the last rule of the table: when matched and processed, it rewrites the central bit of the *dst* vector to 0. Finally, some rules are mutually compatible, e.g. a packet can match the first two

simultaneously. To deal with such cases, rules are assigned priorities. Given a packet, these rules are tried out in decreasing order until one matches, meaning that when a packet can match multiple rules, only the highest-priority one is enforced. In the case of Figure 5.1b, the higher the rule in the table, the higher its priority.



(a) A simple network topology ...

in	*dst*	*src*	*rewrite*	*out*
R1	$10\star$	$01\star$		R2
R1	$1\star\star$	$\star\star\star$		R3
R2	$10\star$	$\star\star\star$		B
R3	$\star\star\star$	$1\star\star$		D
R3	$1\star\star$	$\star\star\star$	$dst[1] := 0$	R2

(b) ... and its configuration

Figure 5.1: Example network

The authors of [Lopes et al., 2015] show how they translate to Datalog reachability from A to B and compute the set of packets that flow this way. Although it is not a requirement to understand the encoding, we invite the reader to try to compute this set by hand, and realize that even such a minimal and extremely simplified problem is actually non-trivial and error-prone. The details of these computations are found, for readability, in Appendix B.

The surpising complexity of this modest example justifies the use of automatic method for actual cases. Figures 5.2 and 5.3 show the Datalog translation of this example. The first figure contains syntactic sugar that encodes the various relevant guard conditions and effects of the routing table, whereas the second uses them to define the actual routing rules.

$$
\begin{array}{lll}
G_{12} & := & dst = 10\star,\ \ src = 01\star\,. \\
G_{13} & := & \neg G_{12}(dst, src),\ \ dst = 1\star\star\,. \\
G_{2B} & := & dst = 10\star\,. \\
G_{3D} & := & src = 1\star\star\,. \\
G_{32} & := & \neg G_{3D}(src),\ \ dst = 1\star\star\,. \\
Id & := & src' = src,\ \ dst' = dst\,. \\
Set0 & := & src' = src,\ \ dst' = dst[2]\ \ 0\ \ dst[0]\,.
\end{array}
$$

Figure 5.2: Encoding routing constraints and effects

Remark 5.1. Figure 5.2 uses a primitive predicate (cf. Section 2.4) denoted as $=$. However, the authors of [Lopes et al., 2015] do not comment on this predicate. On the other hand, they explicitely state that the only fact they put in the EDB is a *symbolic packet*, i.e. $B(\star\star\star\ \star\star\star)$. In that setting, if $x = y$ is the syntactical equality, the third rule of Figure 5.3 can never be used. It should be interpreted as x is compatible and at least as precise as y.

The modelization goes backwards. For example, the third rule of Figure 5.3 can be read as "if a packet with header $dst'\ src'$ can reach B, and it can be specialized as $10\star\ src$, then this specialization can reach R2". Note that the priorities between rules are manually encoded at the level of guards (cf. the second and fifth rules of Figure 5.2).

$$
\begin{array}{l}
R1(dst, src) \leftarrow G_{12}(dst, src),\ \ Id(dst, dst', src, src'),\ \ R2(dst', src')\,. \\
R1(dst, src) \leftarrow G_{13}(dst, src),\ \ Id(dst, dst', src, src'),\ \ R3(dst', src')\,. \\
R2(dst, src) \leftarrow G_{2B}(dst),\ \ Id(dst, dst', src, src'),\ \ B(dst', src')\,. \\
R3(dst, src) \leftarrow G_{3D}(src),\ \ Id, (dst, dst', src, src')\ \ D(dst', src')\,. \\
R3(dst, src) \leftarrow G_{32}(dst, src),\ \ Set0(dst, dst', src, src'),\ \ R2(dst', src')\,. \\
A(dst, src)\ \ \leftarrow R1(dst, src)\,.
\end{array}
$$

Figure 5.3: Datalog translation of the packet flows

Given these rules and an EDB containing the fact $B(\star\star\star\ \star\star\star)$, a Datalog engine can be asked to compute the set of deductible facts about A, i.e. the set of packet headers that, starting from A, can reach B in any form.

5.2.2 Various policies

Although reachability is the backbone of network verification, one may want to check more intricate properties in practice. The authors of [Lopes et al., 2015] introduce the Datalog formalizations of such problems, which we quickly overview.

Example 5.2. (Protection sets) It can be checked that some parts of the network are not reachable by specific elements (e.g., fabric managers are not reachable from guest virtual machines) by looking for a counterexample using the previously introduced method.

Example 5.3. (Reachability sets) Still using the search of a counterexample and the reachability encoding, one can check that all network elements of a given set are reachable from another set (e.g., all fabric managers are reachable from jump boxes (internal management devices)).

Example 5.4. (Equivalence of load balanced paths) In general, traffic from A to B does not follow a single path, but is *balanced* across multiple paths. The way packets are split between the available paths is usually determined by a hash function, which associates an index (within the number of paths) to every packet header.

A more complex question is then to check whether *reachability across load balanced paths is identical regardless of other variables such as hash functions*. This question can also be translated into Datalog, by encoding a hashing scheme as a bit vector h which determines the hashing choices made at every routers, and have a primitive predicate $Select$ that selectively enables a rule given a hashing and a packet header. This $Select$ predicate can then be added as an extra guard to the translation of the routing. Such rules would then look like those of Figure 5.4.

$$
\begin{array}{l}
R2(dst, h) \leftarrow G_{12}(dst, h),\ \ R1(dst, h),\ \ Select(h, dst)\,. \\
R3(dst, h) \leftarrow G_{13}(dst, h),\ \ R1(dst, h),\ \ Select(h, dst)\,.
\end{array}
$$

Figure 5.4: Accounting for load balancing in Datalog formalization

With these new rules in hand, one can issue a query of a node that can receive a packet using one hash assignement, but would not be reached by the same packet under another hash assignement. Such a query, illustrated by Figure 5.5, would then detect inconsistent hashing.

$?A(dst, h_1) \wedge \neg A(dst, h2)$

Figure 5.5: Looking for load balancing inconsistencies

The authors of [Lopes et al., 2015] state that network verification tools of the time, such as [Kazemian et al., 2012, Kazemian et al., 2013, Khurshid et al., 2013], were not able to answer this kind of query.

Example 5.5. (Locality) Another belief one may want to check, is whether, in a data center, traffic within a rack does not leave it. In other words, when a packet in a rack is meant for another place of the same rack, it should not leave the top-of-rack switch. One can easily define a set of places which should not be reached (e.g., using a "forbidden" predicate and a rule for each such place) and check that a packet is meant for a specific subnetwork (e.g., using equality *modulo* a mask). A counterexample can then be queried.

Example 5.6. (Dynamic packet headers) Whereas some network verification tools such as [Kazemian et al., 2013, Kazemian et al., 2012] require *a priori* definitions and implementions to support various protocols, Datalog is flexible enough to only require such work at the level of the definition of the analysis, or even the query.

The authors of [Lopes et al., 2015] illustrate this possibility with an outline of the Datalog encoding of the MPLS protocol [Viswanathan et al., 2001], which relies on label stacking, i.e. the use of packet headers as a stack to store labels as it enters each new *layer* of the protocol, and unstack them when these layers are exited. Datalog does not support terms, and in particular lists, but it can encode within the predicate the number of stacked labels, and use one argument for each.

Remark 5.7. In that sense, the expressivity of Datalog is similar to that of finite automata, where the problem needs to be bounded *a priori* in the absence a dynamic structure such as a stack. The main difference between these two tools in this case are the simplicity and efficiency in which such problems can be formulated and solved. The use of automata for network verification is explored in NetKat (see Section 4.4).

Given a router R, we denote as R^i the forwarding state with a stack of i MPLS labels. This is illustrated by the rules of Figure 5.6, which encode the stacking when going from a router R_5 to R_2. The first rule adds an arbitrary label (2018) when there is none. The second rule expects that a such label is already present and stacks another on top, and so on. The last rule states that the stack overflows when trying to stack more than three labels.

$$
\begin{array}{lll}
R_2^1(dst, src, 2018) & \leftarrow G(dst, src), & R_5^0(dst, src) \\
R_2^2(dst, src, l_1, 2019) & \leftarrow G(dst, src), & R_5^1(dst, src, l_1) \\
R_2^3(dst, src, l_1, l_2, 2020) & \leftarrow G(dst, src), & R_5^2(dst, src, l_1, l_2) \\
Ovlf(dst, src, l_1, l_2, l_3) & \leftarrow G(dst, src), & R_5^3(dst, src, l_1, l_2, l_3)
\end{array}
$$

Figure 5.6: Encoding label stacking in Datalog

Finally, the authors of [Lopes et al., 2015] also mention two subtle bugs they encountered

when working with middleboxes traversal (i.e. ensuring that some class of packets goes some networking component such as a firewall or a load balancer) and the use of backup routers. Both examples require the introduction of too many technicalities to be consistent with the rest of this document, but can be easily and elegantly formalized within Datalog.

Although these various examples illustrate how convenient Datalog can be to model and specify network behaviors, actually verifying them requires the aforementioned programs to scale, which is not the case natively. NoD has been designed with that objective in mind, i.e. with modifications made to the Datalog engine, which are discussed in the next section.

5.3 Modifying a Datalog engine for network verification

The authors of [Lopes et al., 2015] build their tool upon a Z3 implementation of Datalog, called μZ [Hoder et al., 2011]. Their modifications can be split into two main components: a *packing* of two operations in the algebraic preprocessing of the queries, which avoids the very costly representation of the intermediate step, and the choice of data structures for the packets and their rewriting. The first point is not addressed in this document, as it has no impact on our work. The representation of packets, on the other hand, was decisive in the reasoning which led to the developments presented in this thesis.

Relations are one of the key ingredients of Datalog. They are used to model the routers, which, unlike switches, handle some complex network behaviors, such as multicast (a packet gets duplicated into multiple copies which are all sent on their own path) or load balancing. Datalog encodes relations as tables, where each row represents a value tuple.

In the context of network verification, the naive approach is to represent a set of packet headers as a relation, and this relation as a table. However, doing so would result in a table with a number of rows that would be exponential in the number of bits used in the packet headers. For example, assuming we use 128-bit packet headers, accounting for all source addresses that start with a 1 would amount to a table of 2^{127} rows. This approach obviously does not scale.

The authors of [Lopes et al., 2015] mention that they first encoded the Datalog tables using Binary Decision Diagrams [Bryant, 1986]. However, they do not focus on this approach, as they show in [Lopes et al., 2013] that the results are significantly weaker than those of their second representation, which they call *difference of cubes*.

Definition 5.8. A difference of cubes, or DoC, is a set of packet patterns *modulo* exceptions. More precisely, they are of the form

$$\bigcup_i (\nu_i \setminus \bigcup_j \nu_{i,j})$$

where ν_i and ν_j are bit vectors. In the setting of NoD, it can be understood as, for example, "every packet of the form ν_1 except those matching $\nu_{1,1}$ or $\nu_{1,2}$, as well as the packets of the form ν_2 with the exception of those matching $\nu_{2,1}$".

Example 5.9. We do not use the *outer set union*, which requires no illustration, to focus on the exception mechanism. The relation on 4-bit integers

$$***0 \setminus \{*11*, *00*, 1010\}$$

represents the set of naturals that are even (the binary representation must end with a 0), do not have identical "middle bits" (first two exceptions) and different from 6 (hard-coded special case).

The point of this representation lies in the handling of priority among rules. For example, in Figure 5.1, the second rule is used for a packet only if it does not match the first, meaning that the underlying formula for this possibility is of the form $\phi \wedge \neg\psi$. Having a set of exceptions means that this can be used to model the $n + 1^{th}$ rule as

$$\phi \wedge \neg\phi_1 \wedge \cdots \wedge \neg\phi_n$$

where ϕ is the criteria of the rule and ϕ_1 to ϕ_n are those of the first n rules.

Although very efficient in the context of NoD, where the Datalog programs are tailored for each analyzed network, this representation does not always fare well with more generic, abstract programs, as discussed in the next Chapter.

Chapter 6

Octant

> Lorsqu'elle est pratiquée dans les règles de l'art, la prospective permet de repérer les principales métamorphoses qui couvent à bas bruit dans la société avant qu'elles ne s'expriment au grand jour, ce qui nous permet d'anticiper les grandes évolutions à venir.
>
> Jean-Philippe Toussaint, *Les émotions*

Octant is a tool to formalize virtual network models and implementation in Datalog and check properties of those formalizations. It is based on the previously introduced Network Optimized Datalog engine. We justify the need for tools like Octant and show some examples of network and policy formalizations in the context of Openstack in Section 6.1, and discuss in Section 6.2 how the generic description of networking mechanisms triggers a state space explosion issue in NoD.

6.1 A higher-level Datalog model

Network Optimized Datalog, introduced in Chapter 5, allows the specification and efficient verification of network behaviors. However, the Datalog programs have to be tailored for each network and belief to check (see the example of Section 5.2.1, where the rules of the forwarding table are hard-coded into the program). Scaling this proof of concept result as it is to real-world, industrial level would then require either a kind of expertise that is not common amongst network engineers and architects, or a formally introduced and justified program transformation process. Moreover, such specialized programs are more complicated to write, understand and maintain than we could expect or require.

Octant works on networks that were deployed using the OpenStack [Sefraoui et al., 2012] cloud computing platform. In that setting, Octant can fetch the network's configuration and service databases through the REST APIs of the relevant services, i.e. mainly Neutron. This way, the specificities of such analyzed network can be abstracted away at the Datalog level, resulting in simpler and more generic programs.

OpenStack's back-end is based on relational databases, which makes this modelization process relatively straightforward. The two following examples, displaying the Octant detection of multi-attachment and of (a simplified version of) reachability (which does not handle security groups and firewalls), highlight the higher-level nature of this tool w.r.t. NoD.

In both examples, we consider the following extensional predicates: $server(id)$, $router(id)$, $network(id)$, which denote the existence of an homonymous device associated to the provided id, and $port(id, net\_id, device\_id, ip)$, that relates a port to network and a server or router. OpenStack uses many options to define networks, meaning that Neutron tables usually have many columns themselves. Arguments of extensional predicates are then named explicitly, so that those not used in a rule can be omitted (see the program of Figure 6.1, where two or one arguments of the *port* predicate are associated to a variable, depending on the rule). Note that this also helps to quickly distinguish extensional and intensional predicates.

Example 6.1. The program of Figure 6.1 detects virtual machines connected to a network through a chain of routers and other networks. The *linked* predicate is defined as the set of pairs of ports appearing in the same router. Then, *cnt* (connectivity) is akin to the graph reachability specification of Example 2.9. Finally, we say that a server X is connected to a network Z when it contains a network Y which is connected to Z.

$$
\begin{array}{ll}
linked(X,Y) & \leftarrow port(net\_id = X, device\_id = T),\ \ router(id = T), \\
 & \quad\ port(net\_id = Y, device\_id = T). \\
 & \\
cnt(X,X) & \leftarrow port(net\_id = X). \\
 & \\
cnt(X,Z) & \leftarrow linked(X,Y),\ \ cnt(Y,Z). \\
 & \\
cntVM(X,Z) & \leftarrow server(id = X),\ \ port(net\_id = Y, device\_id = X),\ \ cnt(Y,Z).
\end{array}
$$

Figure 6.1: Connectivity in Datalog

This program fragment can be used as a basis to check that a virtual machine does not have a double attachment on networks with different security levels. For example, given the EDB shown in figure 6.2, we can deduce $cntVM('M1','test')$ and $cntVM('M1','prod')$.

$$
\begin{array}{l}
port(id =' p1', device\_id =' M1', net\_id =' test'). \\
port(id =' p2', device\_id =' M1', net\_id =' inter'). \\
port(id =' p3', device\_id =' R1', net\_id =' inter'). \\
port(id =' p4', device\_id =' R1', net\_id =' prod'). \\
server(id =' M1'). \\
router(id =' R1').
\end{array}
$$

Figure 6.2: Sample EDB

The rule of Figure 6.3 will then be able to detect the multi-attachment of $M1$.

$$
doubleAttach(X) \leftarrow cnt(X,Y),\ \ cnt(X,Z),\ \ not\ \ Y = Z.
$$

Figure 6.3: Catching multi-attachment

Example 6.2. To check if traffic can actually reach a given virtual machine, we need to model forwarding rules in routers and network appliances (security groups, firewalls) along the paths. Routing in network also relies on subnet masks, i.e. bitmasks that are used with a bitwise AND to project IP addresses to the relevant slice. For example, address 111.112.113.114 *masked* with 255.255.255.0 returns the 111.112.113.0 prefix.

Let $route(id, router\_id, prefix, mask, port\_id)$ represent an explicit rule on a router that identifies every packet whose destination attribute matches $prefix$ projected over $mask$, and sends them to the router port denoted by $port\_id$. Figure 6.4 extends the *linked* predicate for a given IP address. The & and > primitive predicates stand respectively for the bit-wise AND and comparison on bit vectors.

$$
\begin{array}{l}
linked(X, Y, IP) \leftarrow \\
\quad port(net\_id = X,\ \ device\_id = T), \\
\quad router(id = T), \\
\quad match\_route(T,\ \ M,\ \ IP,\ \ P), \\
\quad not\ \ better\_route(T,\ \ IP,\ \ M), \\
\quad port(id = P,\ \ net\_id = Y,\ \ device\_id = T). \\
\\
match\_route(T,\ \ M,\ \ IP,\ \ P) \leftarrow \\
\quad router(router\_id = T,\ \ prefix = S,\ \ mask = M,\ \ port = P), \\
\quad IP\ \ \&\ \ M = S. \\
\\
better\_route(T, IP, M) \leftarrow match\_route(T, M2, IP, P),\ \ M2 > M.
\end{array}
$$

Figure 6.4: Simple network reachability in Datalog

This program fragment represents a typical use of a rule selection mechanism, as the *linked* predicate requires not only that the selected route matches ($match\_route$), but also that it has the highest priority possible (or, equivalently, that there exists no matching higher-priority rule, cf. $not\ better\_route$). In this context, priority is indexed on the length of the mask, i.e. the longest matching routing rule is selected.

Even though it is omitted for clarity and conciseness, the security layer, i.e. firewalls and security groups, is also modelized in Octant. Just like the core of forwarding, its specification is both generic and abstract. Overall, Examples 6.1 and 6.2 display much higher-level specifications than what is done within NoD, for example in Section 5.2.1.

We emphasize that NoD and Octant, although closely related, do not address the same problems. A NoD program checks properties over a given network by mixing the descriptions of these general network properties (e.g. accessibility) and the specificities of the given network (e.g. topology, forwarding tables). On the other hand, an Octant program abstracts the aforementioned specificities to focus on the description of network properties, which can then be checked against a variety of concrete networks. Although highly beneficial, this *lift* in abstraction and clarity comes at a cost, which is discussed in the next section.

6.2 The cost of genericity

Octant is executed using Network Optimized Datalog [Lopes et al., 2015], a choice made for its efficency with the representation of routing rules and packet rewriting (see Section 5.3 of this document). Natively, NoD is simply not able to execute a program such as the one seen in Figure 6.4, as it does not come with DoC representations for primitives such as <. Section 6.2.1 first discusses the difficulties with the implementation of said primitive in DoC, then Section 6.2.2 explains how these difficulties *blow up* with the use of negation in Octant programs – two factors that would probably be harmless taken separately, but become a significant bottleneck when in conjunction.

6.2.1 Need and implementation of generic primitives

Handling in practice the new level of abstraction brought by Octant introduces the need for the addition and efficient implementation of primitive predicates, such as bitwise conjunction (&), equality and comparison, which all appear in Figure 6.4. NoD encodes everything using the *difference of cubes* (DoC) representation, introduced in Section 5.3. Many usual primitives, such as the comparison with a constant, equality and bitwise operations can be encoded very efficiently using DoCs, as illustrated by the following examples.

Example 6.3. Given a variable v typed as a four-bit integer, the $v \geq 1101$ comparison can be encoded in DoCs as

$$\star\star\star\star \setminus \{0\star\star\star, 10\star\star, 1100\}$$

In general, any comparison of a variable and a constant will be encoded by a similar mechanism, where a linear number of prefixes are enough to eliminate any irrelevant value.

Example 6.4. Given two variables v_1 and v_2 representing four-bit integers, the equality relation $v_1 = v_2$ can be encoded linearly in DoCs by forbidding the existence of a dissonant pair of values, as illustrated by the following eight-bit DoC:

$$\underbrace{\star\star\star\star}_{v_1}\ \underbrace{\star\star\star\star}_{v_2} \setminus \{\underbrace{1\star\star\star}_{v_1}\underbrace{0\star\star\star}_{v_2},\ \underbrace{0\star\star\star}_{v_1}\underbrace{1\star\star\star}_{v_2},\ \underbrace{\star1\star\star}_{v_1}\underbrace{\star0\star\star}_{v_2}, ...\}$$

Example 6.5. The encoding of bitwise operations is slightly more twisted but linear as well. The idea is simply to harness the binary nature of bits and forbid the opposite of the given operation's logical table, as illustrated by the DoC formalization of bitwise conjunction shown in Figure 6.5.

$$\begin{array}{lcccccccccccccc}
 & & v_1 & v_2 & v_1\&v_2 & & v_1 & v_2 & v_1\&v_2 & & v_1 & v_2 & v_1\&v_2 & & v_1 & v_2 & v_1\&v_2 & \\
\star\star\star\star & \setminus & \{\ 0\star\star\star & 0\star\star\star & 1\star\star\star, & & \star0\star\star & \star0\star\star & \star1\star\star, & & \star\star0\star & \star\star0\star & \star\star1\star, & & \star\star\star0 & \star\star\star0 & \star\star\star1, & \\
\star\star\star\star & & 0\star\star\star & 1\star\star\star & 1\star\star\star, & & \star0\star\star & \star1\star\star & \star1\star\star, & & \star\star0\star & \star\star1\star & \star\star1\star, & & \star\star\star0 & \star\star\star1 & \star\star\star1, & \\
\star\star\star\star & & 1\star\star\star & 0\star\star\star & 1\star\star\star, & & \star1\star\star & \star0\star\star & \star1\star\star, & & \star\star1\star & \star\star0\star & \star\star1\star, & & \star\star\star1 & \star\star\star0 & \star\star\star1, & \\
 & & 1\star\star\star & 1\star\star\star & 0\star\star\star, & & \star1\star\star & \star1\star\star & \star0\star\star, & & \star\star1\star & \star\star1\star & \star\star0\star, & & \star\star\star1 & \star\star\star1 & \star\star\star0 & \}
\end{array}$$

Figure 6.5: Inverse truth table in differences of cubes

We use twelve-bit vectors, which represent the two values of the two variables, and the result of the bitwise conjunction, on four bits each. This difference of cube uses the binary nature of bits to encode the conjunction by specifying what should not be produced, e.g. two 0 at the same index in v_1 and v_2, and a 1 at the same index in the result.

However, not every primitive can be handled in an efficient way with the difference of cubes representation.

Example 6.6. When representing the relation $v_1 \geq v_2$, we need to compare prefixes rather than isolated bits, as illustrated by Figure 6.6. This means that the tricks seen in the previous examples do not apply to this case, and end up with a representation that is exponential in the number of bits the integers use.

$$\underbrace{\star\star\star\star}_{v_1}\ \underbrace{\star\star\star\star}_{v_2} \setminus \{\underbrace{0\star\star\star}_{v_1}\underbrace{1\star\star\star}_{v_2},\ \underbrace{00\star\star}_{v_1}\underbrace{01\star\star}_{v_2},\ \underbrace{10\star\star}_{v_1}\underbrace{11\star\star}_{v_2},$$
$$\underbrace{000\star}_{v_1}\underbrace{001\star}_{v_2},\ \underbrace{010\star}_{v_1}\underbrace{011\star}_{v_2},\ \underbrace{100\star}_{v_1}\underbrace{101\star}_{v_2},\ \underbrace{110\star}_{v_1}\underbrace{111\star}_{v_2}...\}$$

Figure 6.6: Encoding the $v_1 \geq v_2$ relation

As seen in Figure 6.4, optimality is a key component of the definition of forwarding, which itself is a building block of the other components. However, optimality can only be expressed using comparisons, whose representations using differences of cubes are exponential. One might expect that this exponential representation[1] is not really a problem in practice, as we intuitively do no want to compute the full set of $v_1 < v_2$ pairs in Octant, but rather the set of addresses which are greaters than the one corresponding to the given M variable.

6.2.2 Effects of the use of negation

As discussed in Section 2.3, negation in Datalog is dealt with by stratifying the program and saturating each intermediate stratum. Having a full and complete definition of what is true being the requirement to define something as false.

This is also the mechanism used in μZ, as indicated in Section 2.2 of [Hoder et al., 2011]. This means that, when executing the forwarding program of Figure 6.4, the rule defining *linked* requires the whole table of *better_route* to be computed. Combined with the inefficient representation of the comparison between two variables, the overall result is extremely inefficient and unusable in practice.

However, as intuited above, the mechanism in practice does not match the behavior that one might first expect without knowing the handling of negation in Datalog. Indeed, from a more operational point a view, the *better_route* rule is used whenever we need to check that the currently considered route is not surpassed in priority by another.

More concretely, seeing Datalog as a more traditional programming language with *functions* that are *called* in a top-down fashion, one may consider that the M parameter in the *better_route* rule is fully defined at run-time, which does not match the bottom-up, stratified bevahior of μZ. The core of this document, and the work it represents, was then to introduce optimizations which provide clues to the Datalog engine and, somehow, help it simulate a top-down behavior to avoid the performance caveat just described. This is the topic of Part V, whereas Part IV first extends the tools at our disposal.

[1] The program of Figure 6.4 uses a strict comparison whereas it was broad in Example 6.6. That is because $<$ is even harder to directly implement and is easier described as the negation of $\geq$.

Part IV
Extension of tools

Some of the tools previously introduced have been extended for the purpose of our work. This part introduce some new sequence and tree finTypes that we built upon the Mathematical Components Coq library, and then present the Datalog trace semantics we developed, with both the paper and (certified) Coq definitions.

Chapter 7

New sequence and tree finTypes

> On ne peut pas entrer deux fois dans le même fleuve
>
> Héraclite, *Fragments recomposés*, traduit du grec ancien par Marcel Conche

Mathematical Components (MathComp) is a Coq library that contains types and tools to define and formally prove traditional, pen and paper mathematical results within Coq. It notably contains a hierarchy of types with the following properties: having decidable equality (`eqType`), having a choice function (`choiceType`), being countable (`countType`), and finiteness (`finType`). These types are properly introduced in Section 3.1.

Our main focus here is `finType`, of which DatalogCert [Benzaken et al., 2017b, Dumbrava, 2016] already made an extensive use (see Chapter 3). Some of our additions could not fit in the different structures provided by `finType`, meaning that we had to develop our own sequence and tree finite types. We first introduce these new types, and then present some changes we make to DatalogCert. The uses of the tree `finType`s will be shown when discussing the relevant definitions, i.e. in Sections 8.3 and 10.4.4.

7.1 Bounding sequences

We first introduce two types of sequences, the finiteness of which comes from syntactic or semantic criteria.

7.1.1 Syntactically bounded sequences

MathComp already contains a type for sequences of exactly a given length, called `tuple`, which inherits the finiteness property of any finType used for the elements. We introduce `Wlist`, the type of lists (sequence, or `seq`, in MathComp's nomenclature) bounded by a given `nat`.

The definition of `Wlist` can be found in Figure 7.1. Unlike `tuple`, it is not implemented using a signature type, but an inductive. Its definition follows that of traditional lists, with the addition of a bound of the number of elements within the type. The *empty list* case introduces any bound, meaning that precision can be lost at this level, whereas the addition of an element simply increments the bound.

```
Inductive Wlist (X: Type): nat -> Type :=
  Bnil  : forall n, (Wlist X n)
| Bcons : forall n, X -> (Wlist X n) -> (Wlist X n.+1).
```

Figure 7.1: Definition of syntactically bounded sequences

The finiteness of `Wlist A n`, where `A` is a finType and `n` a `nat`, is shown via induction. Figure 7.2 displays the base case of the proof, i.e. showing that `Wlist A 0` – which only contains `Bnil 0` – is equivalent to the `unit` type.

```
Definition g (_: unit) := (Bnil A 0).

Definition f (x: Wlist A 0) := tt.

Lemma nil0 : forall (x: Wlist A 0), x = (Bnil A 0).

(* forall x. g (f x) = x *)
Lemma cancelfg : cancel f g.
```

Figure 7.2: Base case for the finiteness proof of Wlist

Figure 7.3 shows the inductive case, in which an element of type `Wlist A n+1` is simply transformed as an element of type `unit + (Wlist A n * A)`. The right side of the sum type is the *normal* situation, where the original element was the result of the addition of something to the list using `Bcons`. On the other hand, a `Wlist A n+1` may also be an empty list, produced by `Bnil`, hence the presence of the `unit` type in the sum.

```
Definition gg (n: nat) (x: unit + A * Wlist A n) : (Wlist A n.+1) :=
match x with
| inl _ => Bnil A n.+1
| inr (a, l) => Bcons a l
end.

Derive Signature for Wlist.

Equations ff (n : nat) (x : Wlist A n.+1) : (unit + A * Wlist A n) :=
  ff (Bnil _) := inl tt ;
  ff (Bcons a l) := inr (a,l).

Lemma cancelffgg: forall n, cancel (@ff n) (@gg n).
```

Figure 7.3: Inductive case for the finiteness proof of Wlist

Notation 7.1. In MathComp, `S n` is usually denoted as `n.+1`.

Remark 7.2. The *straightforward* implementation of `ff` failed to deal with the `n` argument without adding `return unit + A * Wlist A (pred n)` to the match. Before thinking about this addition, we have been advised to use the Equations library [Sozeau, 2010], which does work and remained in the code.

The sum or product of two `finTypes` is a `finType` itself, and the finiteness of `Wlist A n` is the induction hypothesis. Figure 7.4 shows how these facts are combined to end the proof and fit `Wlist` within the `finType` framework, or *mixin* (see Definition 3.8).

```
Variable A: finType.

Definition wlist0_finMixin :=
   @CanFinMixin (wlist0_countType A) unit_finType (@f A) (@g A) (@cancelfg A).

Definition wlist0_finType := FinType (wlist0_countType A) wlist0_finMixin.

Fixpoint wlistn_finMixin (n:nat): Finite.mixin_of (wlistn_countType A n).
elim n.
rewrite cteq. (* wlistn_countType 0 = wlist0_countType *)
exact wlist0_finMixin.
intros n0 EF.
apply (@CanFinMixin
          (wlistn_countType A n0.+1)
          (sum_finType unit_finType
                       (prod_finType A (FinType (wlistn_countType A n0) EF)))
          (@ff A n0) (@gg A n0) (@cancelffgg A n0)).
Defined.

Definition wlistn_finType n :=
   Eval hnf in (@FinType (wlistn_choiceType A n) (wlistn_finMixin n)).
```

Figure 7.4: Wrapping-up the finiteness proof of Wlist

Remark 7.3. The proof is shorter and simpler than the one for the finiteness of `tuple`. This probably stems from the fact that it is less precise, as we do not explicitely state or prove the cardinal of `Wlist`.

Using this type in practice can be cumbersome, as adding an element to a `Wlist` bounded by nat `m` returns an element of type `Wlist m+1`, even though there may actually be much less than `m+1` items in the list. We wrote functions that map elements of type `Wlist` to usual sequences, and the other way around. In the second case, elements from the list can be lost if it was too long for the bound of the returned `Wlist`.

However, we also wrote cancellation lemmas, shown in Figure 7.5, relying on properties about a given sequence's length. This controlled back and forth allowed us to use `Wlist` in practice, at the reasonable cost of tracking the size of the studied lists explicitly.

```
Lemma wlist_seqK (l : Wlist X m) : (seq_to_wlist m (wlist_to_seq l)) = l.

Lemma seq_wlistK (l : seq X) (H : size l <= m) :
   (wlist_to_seq (seq_to_wlist m l)) = l.
```

Figure 7.5: Relating sequences and Wlists

Remark 7.4. In retrospect, this type would have been much more usable in practice had it been defined as a signature type, such as the one of Figure 7.6.

```
Variable A : Type.
Structure Wlist (w : nat) := {ws :> seq A; Hw : size ws <= w}.
```

Figure 7.6: Signature version of Wlist

We used `Inductive` instead as it allowed us to rely on traditional proof methods. Showing the finiteness of such a signature type would be similar to what is done for the next type.

7.1.2 Sequences bounded by unicity

The second type of bounded sequences we introduce is a signature type called `uniq_seq`, shown in Figure 7.7. It uses the `uniq` predicate already defined in MathComp to circumscribe the `seq` type to lists that do not contain the same element multiple times.

```
Fixpoint uniq s := if s is x :: s' then (x \notin s') && uniq s' else true.
Structure uniq_seq {A : eqType} := {useq :> seq A ; buniq : uniq useq}.
```

Figure 7.7: Definition of lists with unicity

To prove the finiteness of `uniq_seq` over a finite type `A`, it is injected into a tuple of length bounded by `#|A|`, i.e. the cardinal of `A`. The injection is shown in Figure 7.8.

```
Lemma size_useq {A : finType} (d : @uniq_seq A) : size d < #|A|.+1.

Definition tag_of_uniq_seq {A : finType} (d : @uniq_seq A)
                           : {k : 'I_#|A|.+1 & k.-tuple A} :=
  @Tagged _ (Sub (size d) (size_useq d))
            (fun k : 'I_#|A|.+1 => k.-tuple A) (in_tuple d).
```

Figure 7.8: Injecting uniq_seq into tuples

Tuples over a `finType` being finite types themselves, this injection shows that any `uniq_seq` over a `finType` `A` is as well. Figure 7.9 displays the technicalities for MathComp enthusiasts.

```
Definition uniq_seq_of_tag {A : finType} (t : {k : 'I_#|A|.+1 & k.-tuple A})
                           : option (@uniq_seq_countType A) :=
  insub (val (tagged t)).

Lemma tag_of_dbranchK {A : finType} :
   pcancel (@tag_of_uniq_seq A) uniq_seq_of_tag.

Definition uniq_seq_finMixin {A : finType} :=
   PcanFinMixin (@tag_of_dbranchK A).

Canonical uniq_seq_finType {A : finType} :=
   Eval hnf in FinType (@uniq_seq A) uniq_seq_finMixin.
```

Figure 7.9: Finiteness of uniq_seq

We provide two functions that add an element `x` at the head of an `uniq_seq l`. The first, shown in Figure 7.10, requires a proof of the absence of the element in the list, – `x \notin l` – to build a new proof of unicity for the result – `uniq (x::l)`.

```
Lemma andP_to_uniq {A : eqType} {t : A} {b : uniq_seq}
                   (H : t \notin (useq b) /\ uniq b) :
   uniq (t::b).

Definition ucons {A : eqType} (t : A) (b : uniq_seq)
                 (H : t \notin (useq b)) : uniq_seq :=
   {| useq := t :: b; buniq := (andP_to_uniq (conj H (buniq b))) |}.
```

Figure 7.10: Adding an element to a uniq_seq

The `\notin` predicate being of boolean type, i.e. decidable, the second version of the function simply checks whether the given element is absent from the list and, if it is the case, extracts the relevant proof to build the new, enriched `uniq_seq`. If the element was already in the list, the latter is simply returned, as shown in Figure 7.11.

```
Definition pucons {A : eqType} (t : A) (b : @uniq_seq A) : @uniq_seq A :=
 match Sumbool.sumbool_of_bool (t \notin (useq b)) with
    | left H => ucons H
    | in_right => b end.
```

Figure 7.11: Trying to add an element to a uniq_seq

Finally, we also provide in Figure 7.12 a definition of the empty list seen as an `uniq_seq`.

```
Definition unil {A : eqType} : @uniq_seq A  :=
  {| useq := [::]; buniq := is_true_true |}.
```

Figure 7.12: Empty sequence with unicity

7.2 Bounding trees

The second family of types we introduce are trees. Like the sequence types of Section 7.1, there are two finite tree types, one of them being bounded purely syntactically, whereas the other partially relies on a semantics criterion. Also note that, in this instance, the former is used as a backbone to the latter.

7.2.1 Generic trees in MathComp

MathComp contains an unbounded, variable-arity tree type, called `tree`, which is shown to be a `countType`. However, the nodes of this type can only contain `nat` elements[1], whereas we need to use other types for our work. If these types are shown to be countable themselves,

[1] For no particuliar reason, according to a member of the MathComp team.

we could use the `nat` encoding of their elements, but that seemed like an unnecessary layer of technicality. We then start by introducing our own generic, unbounded variable-arity tree type, called `ABtree`, show in Figure 7.13.

```
Inductive ABtree: Type :=
  ABLeaf : B -> ABtree
| ABNode : A -> seq ABtree -> ABtree.
```

Figure 7.13: Generic tree type

We needed to develop several functions around this type. A core tool to reason about any recursive type is an induction principle, but the principle automatically generated by Coq did not handle properly the sequence of descendents, so we introduced ours, the boolean version of which is shown in Figure 7.14. The base case scenario requires that the given property is enforced by leaves, whereas recursion ensures that a property enforced by a list of trees will be preserved when the same list is used as the descendents of a new node.

```
Lemma abtree_ind : forall (P : ABtree -> bool),
   (forall x : B, P (ABLeaf x))
-> (forall h : A, forall l : (seq ABtree),
     ((all P l) -> P (ABNode h l)))
-> forall t : ABtree, P t.
```

Figure 7.14: Induction principle for generic trees

A first definition we require later (cf. Section 7.2.3) is tree membership. Figure 7.15 shows its definition as a boolean predicate. Note that having the returned type as a boolean implies the use of the decidable equality `==`, which in turn requires the node type `A` to be defined at least as an `eqType`.

Notation 7.5. The `~~` notation is a boolean negation defined in MathComp (and not a double propositional one).

Notation 7.6. Predicates `all` and `has`, which were formally introduced in Definition 3.11, are roughly the decidable versions of `forall` and `exists` for sequences over `eqTypes`.

```
Fixpoint ABin {A : eqType} {B : Type} (x : A)
              (t : @ABtree A B) : bool :=
  match t with
    | ABLeaf _ => false
    | ABNode y l => (x == y) || (has (ABin x) l) end.

Definition ABnotin {A : eqType} {B : Type} (x : A)
                   (t : @ABtree A B) : bool := ~~ ABin x t.
```

Figure 7.15: Generic tree membership

Another notion that comes into play in this document (cf. Section 8.3.2) is that of subtrees,

which is defined in Figure 7.16. The `strict_subtree` predicate does not accept cases where the two given subtrees are equal.

```
Fixpoint subtree {A B : eqType} (t1 t2 : @ABtree A B) : bool :=
  match t2 with
    | ABLeaf _ => t1 == t2
    | ABNode y l => (t1 == t2) || has (subtree t1) l end.

Definition strict_subtree {A B : eqType} (t1 t2 : @ABtree A B) : bool :=
  match t2 with
    | ABLeaf _ => false
    | ABNode y l => has (subtree t1) l end.
```

Figure 7.16: Implementing the notion of (strict) subtree

Our end goal is the definition of a tree type where the finiteness comes from a bound on the number of descendants (*syntactic width bound*) and unicity across branches (*semantic height bound*). This type will be defined as a signature type over `ABtree`, but we first introduce another `finType`, which will be used in the finiteness proof.

7.2.2 Syntactically bounded trees

We define in Figure 7.17 the type of trees with bounded height and number of successors, called `Htree`. The bound on the number of successors `w` is enforced using the `Wlist` type, whereas the bound on the height `n` is incremented each time a root is added.

```
(* Max width of the trees *)
Variable w: nat.

Inductive Htree: nat -> Type :=
  BLeaf : forall n, B -> (Htree n)
| BNode : forall n, A -> (Wlist (Htree n) w) -> (Htree n.+1).
```

Figure 7.17: Definition of syntactically bounded trees

```
Lemma leaf0 : forall (x: Htree 0), { y & x=(BLeaf 0 y)}.

Definition fl_aux n (x: (Htree n)): (n = 0) -> B.
case x; [   intros n0 x0 H; exact x0
          | intros; contradict H; auto ].

Definition fl x := (@fl_aux 0 x (@refl_equal nat 0)).
```

Figure 7.18: From leaves to actual objects

The finiteness is shown similarly to that of `Wlist`, with an induction on the height (the finiteness of the *width bounding* is already encapsulated in `Wlist`). In the base case, we show that

trees with height 0, i.e. leaves, are equivalent to the leaf type `B`. We first introduce in Figure 7.18 the transformation from trees to `B`. Figure 7.19 then shows the reverse transformation, and the cancellation lemma.

```
Definition gl (x: B) := (BLeaf 0 x).

Lemma cancelflgl : cancel fl gl.
```

Figure 7.19: Equivalence between Htree of height 0 and leaves

The cancellation lemma can then be used to prove the finiteness of `Htrees` of height 0, as shown in Figure 7.20.

```
Definition htree0_finMixin :=
   @CanFinMixin (htree0_countType A B) B (@fl A B) (@gl A B) (@cancelflgl A B).

Definition htree0_finType :=
   FinType (htree0_countType A B) htree0_finMixin.
```

Figure 7.20: Base case for the finiteness proof of Htree

For the recursive case, we decompose an element of type `Htree n.+1` into an element of `B` if the tree was a leaf, or an element of `A` (the node) and a `Wlist` of `Htree n` (the descendants). The reconstruction is done in the same way and cancellation is also proved, as shown in Figure 7.21.

```
Definition ffl_aux n m (x: Htree m): (m = n.+1) -> B + (A * (Wlist (Htree n) w)).
case x; [
  intros n0 wit E; exact (inl wit)
| intros n0 hd tl E; rewrite <- (eq_add_S _ _ E); exact (inr (hd, tl))].

Definition ffl n (x: Htree (n.+1)) : B + (A * (Wlist (Htree n) w)) :=
   (@ffl_aux n (n.+1)) x (@refl_equal nat n.+1).


Definition ggl n (x: B + (A * (Wlist (Htree n) w))) : (Htree n.+1) :=
  match x with
  | inl y => (BLeaf n.+1 y)
  | inr p => (BNode (fst p) (snd p))
end.

Lemma cancel_fflggl : forall n, (cancel (@ffl n) (@ggl n)).
```

Figure 7.21: Inductive case for the finiteness proof of Htree

Finiteness is preserved by `Wlist` (cf. Figure 7.4), as well as the type product and sum. Assuming finite types `A` and `B`, we can then propagate their finiteness with the induction, as shown in Figure 7.22.

```
Fixpoint htreen_finMixin (n:nat): Finite.mixin_of (htreen_countType A B n).
elim n.
rewrite cteql. (* htreen_countType 0 = htree0_countType. *)
exact htree0_finMixin.
intros n0 EF.
apply (@CanFinMixin
          (htreen_countType A B n0.+1)
          (sum_finType B
                       (prod_finType
                         A
                         (wlistn_finType (FinType (htreen_countType A B n0) EF) w)))
          (@ffl A B n0) (@ggl A B n0) (@cancel_fflggl A B n0)).
Defined.

Definition htreen_finType n :=
   Eval hnf in (@FinType (htreen_choiceType A B n) (htreen_finMixin n)).
```

Figure 7.22: Wrapping-up the finiteness proof of Htree

Remark 7.7. Just like `Wlist` (cf. Remark 7.4), this type could have been defined more in MathComp's spirit, i.e. as a signature type over `ABtree`.

7.2.3 Semantically bounded trees using unicity

Having two different syntactic bounds makes the use of `Htree` quite tedious. Bounding the number of successors of nodes was not a problem in our use case, so we only had to deal with the height. Our trick is again to use unicity, to define – as a signature type – trees with at most `n` successors and unicity across paths, i.e. forbidding an element to appear twice in any branch.

Reusing tree membership (cf. Figure 7.15, the definition of unicity across branches in a tree as a boolean predicate – so it can be used in a signature type – is straightforward, as shown in Figure 7.23.

```
Fixpoint ABuniq {A : eqType} {B : Type} (t : @ABtree A B) : bool :=
  match t with
    | ABLeaf _ => true
    | ABNode x l => ((all (ABnotin x) l) && (all ABuniq l)) end.
```

Figure 7.23: Unicity across branches

The unicity property will be used to bound the height of our trees. We also need an *horizontal* bound, for which we define the width of a tree in Figure 7.24.

```
Fixpoint ABwidth {A B : Type} (t : @ABtree A B) : nat :=
  match t with
    | ABLeaf _ => 0
    | ABNode _ l => (foldr maxn (size l) (map ABwidth l)) end.
```

Figure 7.24: Tree width

Unicity and width are finally tied together to define our bounded tree type, called `WUtree` (for "Width and Unicity bounded tree") and shown in Figure 7.25.

```
Definition wu_pred {A : eqType} {B : Type} {w : nat} (t : @ABtree A B) :=
   ((ABuniq t) && (ABwidth t <= w)).

Structure WUtree {A : eqType} {B : Type} (w : nat) :=
   Wht {wht :> @ABtree A B ; Hwht : @wu_pred A B w wht}.
```

Figure 7.25: Definition of trees bounded by width and unicity

It remains to show that `WUtree` is, or can be seen as a finite type. Our strategy is to use `Htree` as a backbone, i.e. show that unicity across branches enforces a height bounded by the cardinal of the node type, meaning that `WUtree A B w` can be injected into `Htree A B w #|A|`.

We do not go after this result directly, but rather demonstrate a more general lemma, stating that a tree with unicity and elements forming a subset of `E` has a height bounded by `|E|`. This allows us to reason about and isolate the "unused element" (the root of the tree) in the recursive part of the proof, whereas a type is set in stone. The formalization of these lemmas in shown in Figure 7.26.

```
Lemma uniq_ab_size {A B : eqType} (t : @ABtree A B) (s : seq A) :
   ABuniq t
-> all (fun x => x \subset s) (ABbranches t)
-> ABheight t <= size #|s|.
```

Figure 7.26: Core lemma in the proof of finiteness of WUtree

Replacing set `E` by the full type `A`, the subset condition stated in `uniq_ab_size` is trivially true, which provides the height bound of any `WUtree`, as shown in Figure 7.27. It can then be shown that `WUtree A B w` is indeed a subtype of `Htree A B w #|A|`, and thus a `finType`.

```
Lemma height_WUtree {A B : finType} {w : nat} (t : @WUtree A B w) :
   ABheight (wht t) < #|A|.+1.
```

Figure 7.27: WUtree have a bounded height

As for `uniq_seq`, we developed two insertion functions. The first one takes as an argument a proof of the absence of the inserted root in the provided subtrees, whereas the other dynamically checks this (boolean) criteria. The code of these functions is omitted here for succinctness but very similar to Figures 7.10 and 7.11.

7.3 Adding new finite structures to DatalogCert

These new types were designed for additions to DatalogCert (namely, a trace semantics and a static analysis), which are subsequently introduced and discussed in this document. However, their development required some changes in the definitions presented in Chapter 3, the main goal being to define Datalog clauses as a `finType`.

The base ingredient of clauses, i.e. the set of atoms, is not defined as a finite type in DatalogCert, meaning that the first step is to encapsulate it within a `finType`. Figure 7.28 shows how we encode them, i.e. triples consisting of number of arguments bounded by `max_ar`, the predicate symbol, and the tuple of arguments. All these types are finite, meaning that such triples are as well.

```
Notation atom_enc :=
   ({x : 'I_(max_ar.+1) & (symtype * x.-tuple term_finType)%type}).
```

Figure 7.28: Atoms as finite types

Lifting finiteness to clauses leverages the trick already used to bound the variable type in DatalogCert (see Example 3.4), in the sense that we introduce a program-specific bound for the lengths of bodies across clauses in the program. Just like `n`, the number of variables in the program, this value is defined abstractly rather than computed. This value can then be used in conjunction with `Wlist` to define clauses as a `finType`, as shown in Figure 7.29.

```
Parameter bn : nat.

Definition tail := wlistn_finType atom_finType bn.

Inductive clause : Type := Clause of atom & tail.
Inductive gclause := GClause of gatom & wlistn_finType gatom_finType bn.
```

Figure 7.29: Defining clauses as a finite type

```
(* head predicate of a clause *)
Definition hsym_cl cl := sym_atom (head_cl cl).

Definition safe_cl_hd cl :=
   predtype (hsym_cl cl) == Idb.

Definition prog_safe_hds p := all safe_cl_hd p.

Definition safe_edb i :=
   [forall ga in i, predtype (sym_gatom ga) == Edb].
```

Figure 7.30: Formalization of the extensional vs. intensional predicates constraint

We make other minor changes to DatalogCert. The most notable one, shown in Figure 7.30, is the formalization and addition of the constraint to Datalog's syntax described in Section

2.1.2, which was originally missing in DatalogCert. As stated in Remark 2.15, this constraint does not change Datalog's expressivity, but is a crucial hypothesis of the static analysis we introduce in Chapter 10.

Chapter 8

A trace semantics for Datalog

- Tu veux être immortel ?
- En tout cas que mon passage laisse une trace à jamais.
- Ouais mais bon... De là à faire n'importe quoi.

Lewis Trondheim, *Les formidables aventures de Lapinot* (tome 4, *Amour & Intérim*)

As stated in Section 1.2, a Datalog program consists of a set of base facts, called the EDB, and a set of Horn clauses, called rules. During the execution of a program, some rules are first used to deduce new facts from the EDB, then the newly enriched set of facts is used to compute new facts, and so on. The semantics of a Datalog program is a set of facts, meaning that the series of rules used in any deduction of one of these facts is lost in the process.

A program can contain multiple rules, the heads of which are atoms built with the same predicate, which implies that a fact can be deduced using different series of rules. If we want to be able to reason on the full deduction process leading to a fact, we need to introduce a richer semantics for Datalog.

Section 8.1 introduces the notion of collecting and trace semantics, and discusses some forms a Datalog trace semantics may take. Then, Sections 8.2 and 8.3 formalize on paper and in Coq respectively the semantics we introduce, as well as its certification, and discuss its use in DatalogCert.

8.1 Trace semantics and Datalog

Whereas a traditional semantics aims at giving a short and (relatively) simple meaning to a long series of computations, which can then be used to define program properties, which in turn will be determined by static analyses. A collecting semantics defines the strongest static property of interest, i.e. a class of static analyses [Hoare, 1978, Cousot, 2002].

Such semantics include transitive closures of a program's transition relation, state or predicate transformers, forward or backward reachability relations and so on [Cousot, 2005]. However, the basic example is that of computation traces, i.e. semantics that not only *contain* the meaning, or result, of a program, but also the computations that led to this result.

Remark 8.1. Although it might be an unorthodox use of the term, we denote this special case of collecting semantics, to which the semantics we introduce belongs, as *trace semantics*.

A basic example of trace semantics in the context of transition systems is a trace that corresponds to the (ordered) sequence of visited states. In the same spirit, a collecting semantics for an imperative language may typically take the form of an ordered sequence of program location / memory state couples. In comparison, the semantics we introduce is closer to what is done for a functional programming language in [Perera et al., 2012], where the trace takes the form of a tree that unrolls the execution of the program.

However, the trace semantics we introduce only provides a partial order of the different steps of the execution. In that sense, it focuses less on the computation itself, but rather more on the logical structure of the deduction. An alternative, more traditional approach would have been to formalize the trace semantics as a transition system where the states are sets of available facts, and the transitions are labeled with rule / substitution pairs. In that setting, the traces would have been paths, the ordering of the program's execution would have been complete, and the *logical structure* would have been recomputable.

The choice of a *lighter* form of trace semantics stems from the use of the traces in the certification of the static analysis we introduce (see Chapter 10.4), where the logical structure is needed. We then chose to go with a formalism in which this structure was explicit.

Finally, we reckon that this idea of is not entirely novel, at least in presentation. For example, [Halevy et al., 2001] uses *derivation trees* (cf. their Figure 1), which look a lot like our traces. What we claim to introduce is rather the formalization and verified mechanization of such structures.

8.2 Definition

Programs tend to be defined and computed in a linear way, meaning that a classical form of trace semantics is a series of states in a transition system representing the program. On the other hand, the order in which the rules of a Datalog program are used and the new facts are deduced is of no importance, semantics-wise, as long as each predicate is saturated.

This is reflected by our trace semantics, which represents computations as trees that *store* the logical structure of the program's execution, i.e. which rules were used to deduce which facts. The high-level idea is that the leaves are the starting points of the deductions, i.e. facts taken from the EDB, whereas internal nodes represent a deduction via a clause and a substitution, both stored in the node as a couple. The ground atoms of the instantiated body of the clause required for the deduction are (recursively) defined as the descendants of the node.

Definition 8.2. (Datalog trace semantics) More formally, a trace t is recursively defined using the following rules.

t	$::=$	$Leaf(ga)$,	where ga is a ground atom
	$\mid$	$Node(\langle C, \nu\rangle, [t])$,	where C is a clause, ν is a substitution, and $[t]$ is a list of elements of some type t

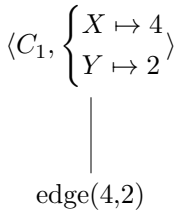
Notation 8.3. In definition 2.32, the set of ground atoms relevant to a program P was

denoted as $\mathbb{B}_P$. In that spirit, the set described above is denoted as $\mathbb{B}_T(P)$.

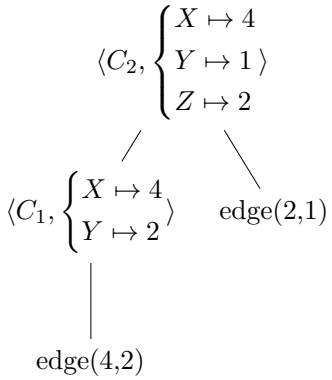
Example 8.4. Consider again the graph reachability program of Example 2.9. Denoting the first and second rules of the program C_1 and C_2, Figure 8.1 displays the construction of the trace corresponding to the deduction of $path(4, 3)$.

More precisely, in Figure 8.1a, the first rule is used with the substitution that maps X to 4 and Y to 2. The instantiated body only contains ground atom $edge(4, 2)$, which is in the EDB of the program, meaning that it can used as a leaf.

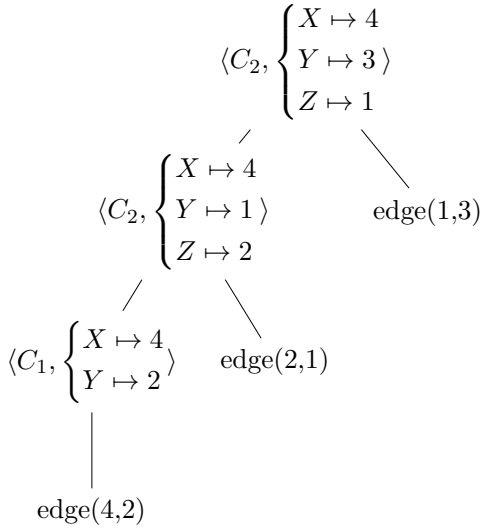
Then, Figure 8.1b shows the deduction of $path(1, 4)$. Using the second rule requires two subtrees, corresponding to deductions of the instantiated two atoms of the tail. The substitution used at the root requires as a first child a trace for the deduction of $path(4, 2)$, so we reuse the previous one. On the right, we need $edge(2, 1)$, which is in the EDB. Finally, Figure 8.1c builds upon Figure 8.1b in a similar manner.



(a) Deduction of $path(4, 2)$



(b) Deduction of $path(4, 1)$



(c) Deduction of $path(4, 3)$

Figure 8.1: Building a Datalog trace

Remark 8.5. Such trees can be seen as proof trees w.r.t. the theory given by a Datalog program and an EDB. The fact that we call them traces stems from our computational approach of Datalog, in the sense that our goal was to study and optimize the execution of Datalog programs, rather than see them as deduction systems.

We can now introduce the various functions that lead to an operational view of this trace semantics. We first need a function that maps a trace to the corresponding deduced fact.

Definition 8.6. (*ded* – **Erasing function from trees to facts)**

$$ded(x) = \begin{cases} f & \text{if } x = Leaf(f) \\ \nu(head(C)) & \text{if } x = Node(\langle C, \nu \rangle, descendants) \end{cases}$$

Example 8.7. The deduced fact in Figure 8.1c is $path(4,3)$, i.e. $ded(t) = path(4,3)$, where t is the trace of the figure.

Our implementation of the trace semantics also *starts* with the initial interpretation, which needs to be seen as a set of traces rather than facts. The *tb* function simply lifts these facts to leaves.

Definition 8.8. (*tb* – **Interpretation to trees)**

$$tb(I) = \{Leaf(f) \mid f \in I\}$$

In the spirit of Section 2.2, the trace semantics is implemented *via* an operator called Tt_P. Also like T_P, it is iterated to build new traces on top of the previously deduced ones. When deducing a new fact with a clause C and a substitution ν, the previous iteration must contain traces for the body of $\nu(C)$. The relation between traces and facts is specified using *ded*.

Definition 8.9. (Tt_P – **Consequence operator on traces)**

$$\begin{aligned} Tt_P(I_t) = I_t \ \cup \ \{&Node(\langle C, \nu \rangle, [F_1, ..., F_n]) \in \mathbb{B}_T(P) \\ &\mid \quad C = A_0 \leftarrow A_1, \cdots, A_n \in P \\ &\wedge \ \forall i \in [1..n], (F_i \in I_t \ \wedge \ ded(F_i) = \nu(A_i))\} \end{aligned}$$

Since this is our own semantics, we need to relate it with the *usual* one, i.e. show its adequacy. Given a program P with interpretation I, this result is expressed in the following lemmas, which are both proved by induction on the number of steps k.

Lemma 8.10. (Datalog trace semantics completeness)
For any number of steps $k, \forall x \in (T_P \uparrow k)(I), \ \exists t \in (Tt_P \uparrow k)(tb(I)), \ ded(t) = x$.
In other words, for any deduction using the fixpoint semantics, the trace semantics contains a tree representing a deduction of the same fact.

Proof. Let $x \in (T_P \uparrow k)(I)$. We need to expose a t in $(Tt_P \uparrow k)(tb(I))$ such that $ded(t) = x$.

If $k = 0$, then $(T_P \uparrow k)(I) = I$ and $(Tt_P \uparrow k)(tb(I)) = tb(I) = \{Leaf(f) \mid f \in I\}$. Since $x \in I$, $tb(I)$ contains $Leaf(x)$. From the definition, $ded(Leaf(x)) = x$.

If $k = n + 1$, then

- $(T_P \uparrow k)(I) = \{head(\iota(C)) \mid C \in P \wedge body(\iota(C)) \subseteq (T_P \uparrow n)(I)\} \cup (T_P \uparrow n)(I)$ (definition)
- $(Tt_P \uparrow k)(tb(I)) = (Tt_P \uparrow n)(tb(I)) \cup \{Node(\langle C, \nu\rangle, [F_1, ..., F_n]) \in \mathbb{B}_T(P)$
  $\mid \quad C = A_0 \leftarrow A_1, \cdots, A_n \in P$
  $\wedge \ \forall i \in [1..n], F_i \in (Tt_P \uparrow n)(tb(I)) \ \wedge \ ded(F_i) = \nu(A_i)\}$
  (definition)
- $\forall y \in (T_P \uparrow n)(I), \exists t \in (Tt_P \uparrow n)(tb(I)),\ ded(t) = y$ (induction hypothesis)

Ground atom x is either on the left or right side of the $\cup$ in the definition of $(T_P \uparrow k)(I)$. In the second case, we apply the induction hypothesis, which provides us a t in $(Tt_P \uparrow n)(tb(I))$ such that $ded(t) = x$. Since t is in $(Tt_P \uparrow n)(tb(I))$, it is also in $(Tt_P \uparrow k)(tb(I))$ (left side of the definition). If x is in the left side of the definition of $(T_P \uparrow k)(I)$, we can extract a clause $C \in P$ and a substitution ι such that

(H1) $x = head(\iota(C))$

(H2) $body(\iota(C)) \subseteq (T_P \uparrow n)(I)$

We now use the right-hand part of the definition of $(Tt_P \uparrow k)(tb(I))$, with C and ν. The condition is true thanks to (H2) and the induction hypothesis, whereas (H1) shows that the tree just built has the right shape.

□

Lemma 8.11. (Datalog trace semantics soundness)
For any number of steps $k, \forall t \in (Tt_P \uparrow k)(tb(I)),\ ded(t) \in (T_P \uparrow k)(I)$.
In other words, any trace corresponds to a deduction of a fact that is actually part of the fixpoint semantics.

Proof. If $k = 0$, then $(T_P \uparrow k)(I) = I$ and $(Tt_P \uparrow k)(tb(I)) = tb(I) = \{Leaf(f) \mid f \in I\}$. Since $t \in tb(I)$, there exists a $f \in I$ such that $t = Leaf(f)$, meaning that $ded(t) = f$.

If $k = n + 1$, then

- $(T_P \uparrow k)(I) = \{head(\iota(C)) \mid C \in P \wedge body(\iota(C)) \subseteq (T_P \uparrow n)(I)\} \cup (T_P \uparrow n)(I)$ (definition)
- $(Tt_P \uparrow k)(tb(I)) = (Tt_P \uparrow n)(tb(I)) \cup \{Node(\langle C, \nu\rangle, [F_1, ..., F_n]) \in \mathbb{B}_T(P)$
  $\mid \quad C = A_0 \leftarrow A_1, \cdots, A_n \in P$
  $\wedge \ \forall i \in [1..n], F_i \in (Tt_P \uparrow n)(tb(I)) \ \wedge \ ded(F_i) = \nu(A_i)\}$
  (definition)
- $\forall t \in (Tt_P \uparrow n)(tb(I)),\ ded(t) \in (T_P \uparrow n)(I)$ (induction hypothesis)

The trace t is either in the left or right part of the definition of $(Tt_P \uparrow k)(tb(I))$. In the first case, we apply the induction hypothesis, which shows that $ded(t)$ is in $(T_P \uparrow n)(I)$, meaning that it is also in $(T_P \uparrow k)(I)$ (left-hand part of the definition).

If t is found on the right side of the definition of $(Tt_P \uparrow k)(tb(I))$, we can extract a clause $C \in P$ and a substitution ν such that

(H1) $C = A_0 \leftarrow A_1, \cdots, A_n \in P$

(H2) $\forall i \in [1..n], F_i \in (Tt_P \uparrow n)(tb(I)) \wedge ded(F_i) = \nu(A_i)$

We now use the left-hand part of the definition of $(T_P \uparrow k)(I)$, with C and ν (for ι). C is in P (H1), and the body of $\nu(C)$ is in $(T_P \uparrow n)(I)$ (induction hypothesis and (H2)). □

Remark 8.12. Section 2.3 recalls how Datalog handles negation, basically by dividing the program into multiple strata which are computed on top one of another. The Tt_P operator can then be used within each stratum, the same way the fixpoint semantics is.

Now that our trace semantics is shown adequate, we can move on to its Coq implementation.

8.3 Coq implementation and certification

This section introduces the core definitions and lemmas of the Coq implementation of our Datalog trace semantics. In particular, we discuss how the finiteness of the trace type impacts the completeness proof shown in Section 8.2.

8.3.1 Definitions

The leaves of the traces are facts, or ground atoms, which are already defined in Figure 3.7. The node elements, which form a type called `rul_gr`, are pairs with a clause and a substitution, as shown in Figure 8.2.

```
Inductive rul_gr := | RS : clause -> sub -> rul_gr.
```

Figure 8.2: Node type for the trace semantics

Remark 8.13. Defining this node type as an Inductive rather than a simple pair is more of a personal taste than an actual need.

```
(** Conversion to and from pair so that we have a cancellable *)
Definition rul_gr_rep l := match l with
  | RS c g => (c, g) end.

Definition rul_gr_pre l := match l with
  | (c, g) => RS c g end.

Lemma rul_gr_repK : cancel rul_gr_rep rul_gr_pre.

Definition rul_gr_finMixin :=
(@CanFinMixin rul_gr_countType (prod_finType clause_finType sub) _ _ rul_gr_repK).

Canonical rul_gr_finType := Eval hnf in FinType rul_gr rul_gr_finMixin.
```

Figure 8.3: Redefining rul_gr as a finite type

Clauses and substitutions being now both defined as finite types, this node type is then simply proved to be a `finType` itself, as shown in Figure 8.3.

As for the actual traces, we want them to form a finType so that the trace semantics can be expressed as a set (of traces rather than ground atoms), similarly to the usual Datalog semantics. To do so, we use the `Wutree` type, as defined in Section 7.2.3. The provided width bound is `bn`, the maximal size for the body of a clause (see Figure 7.29). The use of `Wutree` requires a default value for the leaf type, so we assume one, as shown in Figure 8.4.

```
Variable gat_def : gatom.

Definition trace_sem_trees :=
        (@WUtree_sf rul_gr_finType gatom_finType bn gat_def).
```

Figure 8.4: Definition of the trace type

Functions *ded* and *tb*, which respectively relate a trace to a ground atom and map an interpretation to a set of leaves (cf. Definitions 8.6 and 8.8) are easily translated to Coq, as shown in Figure 8.5.

```
Definition ded def (t : trace_sem_trees) := match (val t) with
  | ABLeaf f => f
  | ABNode (RS (Clause h _) s) _ => gr_atom_def def s h end.

(* my_tst_sub enforces typing *)
Definition base_sem_t (i : interp) : {set trace_sem_trees} :=
  [set my_tst_sub (wu_pred_leaf x) | x in i].
```

Figure 8.5: Coq implementations of *ded* and *tb*

The implementation of the trace semantics still relies on `match_body`, which expects the set of ground atoms deduced so far, meaning that we need the ability to see a set of traces as an interpretation. The `ded` is also lifted to sequences of traces, as we will need to compare the ground atoms represented by a list of traces to instantiated bodies of rules. Both lifts are shown in Figure 8.6.

```
Definition sem_tree_to_inter def (ts : {set trace_sem_trees}) : interp :=
   [set ded def x | x in ts].

Definition ded_sub_equal (def : syntax.constant) (lx : seq trace_sem_trees)
                         (s : sub) (ats : seq atom) : bool :=
  (map (ded def) lx) == (map (gr_atom_def def s) ats).
```

Figure 8.6: Lifting ded to sets and sequences of traces

Our implementation of the trace semantics leverages on a function that takes a set over an option type, e.g. `option A`, and filters out all the `None` elements to return a set over `A`. The implementation of this function, called `pset`, is discussed in Section 12.3.1, but Figure 8.7 shows its specification.

```
Lemma pset_spec {A : finType} (x : A) (s : {set (option A)}) :
   Some x \in s <-> x \in pset s.
```

Figure 8.7: Specification of pset

We can now implement Tt_P. As the T_P operator (see Section 3.3.2.2), the Coq definition is split into two parts. First, `cons_clause_t` computes the deductive part, i.e. the traces that are actually added to the semantics. The function, shown in Figure 8.8 expects a default ground atom `def`, a clause `cl` and a previously deduced set of traces `k`.

```
Definition cons_clause_t def (cl : clause) (k : {set trace_sem_trees})
 : {set trace_sem_trees} :=
  let b := (body_cl cl) in
  let subs := match_body (sem_tree_to_inter def k) b in
  pset [set (wutree_option_fst
              (@wu_pcons_seq rul_gr_finType gatom_finType bn gat_def (RS cl s) lx))
                                        | lx : (size b).-tuple trace_sem_trees,
                                          s  : sub in subs &
                                           (ded_sub_equal def lx s b &&
                                            all (mem k) lx)].
```

Figure 8.8: Computing new traces

We first compute the set of relevant substitutions by applying `match_body` to the body of the clause, with `k` seen as an intepretation. Then, we use MathComp's set notations to iterate over every substitution `s` in `subs` and – using the `tuple` type for finiteness – any sequence of traces `lx` that matches the instantiated body (`ded_sub_equal`). For any such pair, we try to build a tree with root `RS cl s` and `lx` as children. If `RS cl s` was already in a trace of `lx`, `wu_pcons_seq` fails, `None` is returned and filtered out by `pset`. Otherwise, the actual tree is returned and preserved.

The second part of Tt_P, shown in Figure 8.9, is similar to function `fwd_chain` from Figure 3.25, as it applies `cons_clause_t` to every clause of the program and adds the original interpretation `k`, here as a set of trees.

```
Definition fwd_chain_t def (k : {set trace_sem_trees}) : {set trace_sem_trees} :=
  k :|: \bigcup_(cl <- p) cons_clause_t def cl k.
```

Figure 8.9: Forward chain step for the trace semantics

Finally, the `sem_t` function, shown in Figure 8.10 simply iterates Tt_P `m` times, starting with the trace translation of interpretation `i`.

```
Definition sem_t (def : syntax.constant) (m : nat) (i : interp) :=
  iter m (fwd_chain_t def) (base_sem_t i).
```

Figure 8.10: Iterating Tt_P

Now that the trace semantics is fully defined in Coq, we can move on to its formal certification.

8.3.2 Adequacy proofs

The certification of the trace semantics and its use in other proofs led to the development of various technical, very specialized lemmas which are not all listed here for conciceness and clarity. The most important two, shown in Figure 8.11, are about the *reverse-monotonicity*, i.e. the fact that all subtrees of a deduced trace are also part of the trace semantics. Note that the second lemma is about strict subtrees, which means that we can add some precision to the number of iterations needed for the capture of `t1`.

```
Lemma trace_sem_prev_trees nb_iter def init :
   forall (t1 t2 : trace_sem_trees), t2 \in (sem_t def nb_iter init)
-> subtree (val t1) (val t2)
-> t1 \in (sem_t def nb_iter init).

Lemma trace_sem_prev_trees_m1 nb_iter def init :
   forall (t1 t2 : trace_sem_trees), t2 \in (sem_t def nb_iter init)
-> strict_subtree (val t1) (val t2)
-> t1 \in (sem_t def nb_iter.-1 init).
```

Figure 8.11: (Reverse-)Monotonicity of the trace semantics

With such technical results in hand, the adequacy Lemmas 8.10 et 8.11 are simply translated, as seen in Figure 8.12.

```
Lemma trace_sem_completeness nb_iter def i :
   prog_safe p
-> [forall x in (sem p def nb_iter i),
       exists y in (sem_t def nb_iter i), ded def y == x].

Lemma trace_sem_soundness nb_iter def i:
   prog_safe p
-> [forall t in (sem_t def nb_iter i),
       ded def t \in (sem p def nb_iter i)].
```

Figure 8.12: Coq implementation of the trace semantics adequacy

Remark 8.14. The proof of Lemma 8.10 needs to be adjusted, because of the type used to define `trace_sem_trees`. Indeed, we do not need to show that there exists a trace t representing a deduction of the fact x, but that there is a trace t **with unicity** representing a deduction of the fact x. This unicity constraint is not stated explicitely in `trace_sem_completeness`, as it is implied by the use of the `WUtree` type. In that sense, this completeness lemma does not state that *every* deduction is captured by the trace semantics of a program, but that a *sufficient* and *representative* subset is.

Proof. The proof is modified in its final case, i.e. when a fact x has just been deduced using T_P, from which we extract a clause C and a substitution ι. Instead of directly building

a tree with $\langle C, \iota \rangle$ as its root, we need to check that it does not already appear in one of the descendants (provided by the induction hypothesis and hypothesis (H2)). If it does not, we can build the tree and proceed as in the previous proof. Otherwise, we extract the incriminated tree with $\langle C, \iota \rangle$ as its root. Thanks to Lemma `trace_sem_prev_trees` and the monotonicity of Tt_P, it is captured by k iterations of Tt_P. □

Part V
Optimizations

Section 6.2 discussed how the use of some primitive predicates with multiple variables leads to performance issues within the Network Optimized Datalog engine. The core of this thesis is then to introduce program analyses and rewritings that transform a Datalog program into a semantically equivalent one that is free of such constructs.

Chapter 9 introduces a first program transformation that aims at the reduction of variables in a Datalog program by trading them for more rules. This rewriting requires an overapproximation of the behavior of the variables to be removed, which may be provided by the static analysis we introduce in Chapter 10. Then, Chapter 11 presents a second rewriting that reduces the encodings of some predicates within Network Optimized Datalog.

These developments are all implemented and certified in Coq, using the formalization presented in Chapter 3 and the extensions introduced in Part IV. Chapter 12 reflects on this process, in particular discussing the lessons we learned along the way, as well as related works.

Chapter 9

Partial program instantiation

> Pour un probabiliste, c'est un rêveur, il a des yeux verts qui le feraient prendre pour un théoricien des nombres, même s'il porte les cheveux aussi longs qu'un théoricien des jeux, de petites lunettes d'acier trotskisantes de logicien et de vieux T-shirts troués d'algébriste
>
> Hervé Le Tellier, *L'anomalie*

Section 6.2 discussed how seriously – i.e. exponentially – the number of variables in the instances of some primivite predicates impacts the performances of the Network Optimized Datalog engine. This Chapter introduces a first program transformation that aims at the reduction of the number of such variables, by duplicating and partially instantiating clauses.

We first provide the intuition behind this rewriting in Section 9.1, then introduce and justify the paper definitions in Section 9.2. We finally dive into the Coq definitions and certfication in Section 9.3. The rewriting assumes some information about the behavior of the transformed program. The next chapter introduces a static analysis that computes this requirement.

9.1 Intuition

In theory, each iteration of the T_P operator (see Definition 2.45), considers every pair of rule and substitution, and deduces new facts using only pairs that match the available facts. In practice, Datalog engines try to be smarter and more optimal. For example, the Coq formalization of Datalog introduced in Chapter 3 produces the minimal set of relevant substitutions using function `match_body` (see Section 3.3.2.1).

Let us now assume we have an efficient way to statically compute an overapproximation of these substitutions, i.e. a set S of n-tuples of values representing instantiations of a set of variables $V_1, ..., V_n$ that all appear in a single rule. The idea behind our rewriting is to provide these value sets to the engine, roughly saying "you do not have to actually take these variables into account, consider only the value tuples in S".

However, we do not want to actually modify any Datalog engine, as it is a tricky and error prone process. It would also obviously be engine-specific, whereas our goal is to build and validate a method that may ultimately be used on top of existing tools (cf. Remark 12.2). Our idea is then to work at the level of the executed program, i.e. to rewrite it so that the

value overapproximations are passed on as clues.

To do so, a simple and surprisingly effective way is to duplicate the rules, instantiating the targeted variables with the provided value tuples. This both eliminates variables and preserves the semantics as, at any point of the computation, the available facts act as a safeguard, in the sense that the instantiated bodies of the clauses must indeed still be checked against them.

$s(X, Y, Z) \leftarrow q(X), p(X, Y, Z)\,.$

Figure 9.1: Defining s(X,Y, Z)

Example 9.1. Consider the program fragment in Figure 9.1, equipped with the interpretation $\{q(1),\ q(2),\ p(1,3,4),\ p(1,3,5),\ p(1,7,8),\ p(3,4,5)\}$. Figure 9.2 shows the minimal set of substitutions that would be built – for example by function `match_body` – to use the rule in that setting.

$$\{\begin{cases} X \mapsto 1 \\ Y \mapsto 3\,, \\ Z \mapsto 4 \end{cases} \quad \begin{cases} X \mapsto 1 \\ Y \mapsto 3\,, \\ Z \mapsto 5 \end{cases} \quad \begin{cases} X \mapsto 1 \\ Y \mapsto 7\,\} \\ Z \mapsto 8 \end{cases}$$

Figure 9.2: Minimal set of substitutions to compute s

and the semantics amounts to $F = \{s(1,3,4),\ s(1,3,5),\ s(1,7,8)\}$. Assume we want to get rid of variables X and Y, which would then correspond to V_1 and V_2 in the second paragraph of the section. In that setting, the set of instantiations S should contain at least the projections on X and Y of the substitutions given shown in Figure 9.3.

$$\{\begin{cases} X \mapsto 1 \\ Y \mapsto 3\,, \end{cases} \quad \begin{cases} X \mapsto 1 \\ Y \mapsto 7 \end{cases}\}$$

Figure 9.3: Substitutions for s projected over X and Y

Finally, let us assume that S is complete but not correct, in the sense that it contains another partial substitution is not relevant w.r.t. the rule and the provided interpretation, such as a mapping from X to 3 and Y to 4. Figure 9.4 sums up the overapproximation – i.e. set of partial substitutions – we consider.

$$\{\begin{cases} X \mapsto 1 \\ Y \mapsto 3\,, \end{cases} \quad \begin{cases} X \mapsto 1 \\ Y \mapsto 7\,, \end{cases} \quad \begin{cases} X \mapsto 3 \\ Y \mapsto 4 \end{cases}\}$$

Figure 9.4: Overapproximation S of variables X and Y

We can use the S set to replace the rule from Figure 9.1 by the three found in Figure 9.5. The first two rules can be used – in conjunction with the projections on Z of the substitutions of Figure 9.2 – to deduce the facts in F. On the other hand, the third rule can not be used, as $q(3)$ does not match any fact in the EDB, or that will be subsequently deduced. In that sense, although it comes from a strict and incorrect overapproximation, this rule is *harmless*.

$s(1,3,Z) \leftarrow q(1), p(1,3,Z)$.
$s(1,7,Z) \leftarrow q(1), p(1,7,Z)$.
$s(3,4,Z) \leftarrow q(3), p(3,4,Z)$.

Figure 9.5: New, partially instantiated definition of s(X,Y, Z)

Remark 9.2. Example 9.1 is of course particularly simple, in particular as the code snippet used for illustration does not make use of recursion. However, the argument made at the end of the example can be adapted to this setting. The idea is that the first, recursion-free iteration of T_P on the considered rule does not change the deduced *intermediate* set of facts. Then, the second iteration will be based on the right interpretation, thus preserving again the deduced facts, and so on. The next section formalizes both the transformation (including the hypothesis on the provided set of substitutions) and justification.

9.2 Definition and proof

The partial program instantiation first assumes a program P and an initial interpretation – or EDB – I. As stated in the previous section, we may want to focus on a subset of the variables appearing in the program. This set, written $V_1, ..., V_n$ at the beginning of Section 9.1, shall be denoted as R here. Finally, the instantiation requires a set of substitutions S that captures the actual computation for the program w.r.t. the variables in R. To formally define this hypothesis, we first need to define the restriction over a substitution.

Definition 9.3. (Restriction of a substitution) Given a substitution σ and a set of variables X, the restriction of σ over X, written $\sigma_{|X}$, is defined as

$$\sigma_{|X}(x) = \begin{cases} \sigma(x) & \text{if } x \in X \\ x & \text{otherwise} \end{cases}$$

We can now express the completeness condition over the set of substitutions S. The idea is to state that, whenever a clause C contains at least one variable to instantiate, the restriction over the relevant variables of any substitution that can match C after some number of iterations of T_P appears in S.

Since both the *match* relation and T_P operator are monotonic, w.r.t. the given interpretation and number of steps respectively, a substitution matches C after some number of iterations of T_P iff it matches C w.r.t. the full semantics of the program. This fact justifies the following definition of the completeness of S, where $vars(C)$ returns the set of variables that appear in clause C.

Definition 9.4. (Completeness of S)

$$\forall C \in P, |vars(C) \cap R| > 0 \Rightarrow (\forall \nu, match(\nu, C, (T_P \uparrow \omega)(I)) \Rightarrow \nu_{|R} \in S).$$

Remark 9.5. Any method that takes a program and computes a set of substitutions which satisfies the previous criteria, such as the static analysis introduced in the next chapter, can then be used in conjunction with this rewriting, to *fuel* it.

Now that we introduced the context of the rewriting, we can define the actual transformation.

Using function *dom*, that returns the domain of a substitution (see Definition 1.23), any clause can be instantiated using the *inst* function.

Definition 9.6. (*inst* – **clause-level instantiation)**

$$inst(C) = \begin{cases} \{\nu(C) \mid \nu \in S \ \wedge \ dom(\nu) = R \cap vars(body(C))\} \\ \qquad\qquad\qquad\qquad \text{if } |vars(C) \cap R| > 0 \\ \{C\} \qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

Overloading the *inst* notation, the previous definition can naturally be lifted to full programs.

Definition 9.7. (*inst* – **program-level instantiation)**

$$inst(P) = \bigcup_{C \in P} inst(C)$$

Remark 9.8. The $|vars(C) \cap R| > 0$ hypothesis in Definitions 9.4 and 9.6 is rather inelegant, but was required to work with clauses that do not contain any variables of R. An alternative way to deal with them would have been to always manually add the empty substitution to S. Since we did not think this solution was much more satisfactory, we went for the one that felt more efficient in practice. We indeed expect that, in real use cases, many rules will not contain variables to instantiate, meaning that we should quickly take them out of the instantiation process rather than vainly go over the entirety of S for each of them.

Now that the transformation and its hypothesis have been defined, we need to prove that it is semantics-preserving. We first need the following technical result.

Notation 9.9. Given a set X, we denote as $\overline{X}$ the set complement (within the type of the elements). Note that, in particular, for any substitution σ and set of variables V, $\sigma_{|\overline{V}}$ is the restriction of σ to the variables which are not in V.

Lemma 9.10. For any substitution σ and clause C such that σ matches C, then, for any subset of the program variables X, then $\sigma_{|X}$ matches $\sigma_{|\overline{X}}(C)$. In other words, when part of σ is used to instantiate the clause, then the rest of the substitution matches the result.

Proof. First proved at the level of atoms by induction on the arguments, then for clauses using an induction as well, on the body this time. □

The actual adequacy results are broken down into completeness and soundness. Our reference point is the Datalog fixpoint semantics introduced in Section 2.2.2. Intuitively, we show that the same facts are actually deduced in the same number of steps.

Theorem 9.11. (Transformation completeness) For any program P, initial interpretation I and number of steps k, the transformed program deduces in k iterations of T_P every fact that was computed after the same number of steps in the original program, i.e. $(T_P \uparrow k)(I) \subseteq (T_{inst(P)} \uparrow k)(I)$.

Proof. We proceed by induction on the number of steps k. In the base case, the definitions imply that $(T_P \uparrow 0)(I) = (T_{inst(P)} \uparrow 0)(I) = I$, which is even more general than our goal.

In the recursive case, let f be a fact in $(T_P \uparrow k+1)(I)$. Then, f was either already in the previous iteration $(T_P \uparrow k)(I)$, or it has just been deduced and added. In the first

scenario, the induction hypothesis gives us that f is also in $(T_{inst(P)} \uparrow k)(I)$, and thus in $(T_{inst(P)} \uparrow k+1)(I)$ by definition.

In the second scenario, we can extract a clause C from P and a substitution ν such that ν matches C w.r.t. $(T_P \uparrow k)(I)$ and f is the head of $\nu(C)$. If C has no relevant variable, i.e. $|vars(C) \cap R| = 0$, we reuse the same clause and substitution. The atoms in the body of the instantiated clause being in $(T_{inst(P)} \uparrow k)(I)$ (induction hypothesis), we can indeed deduce f in $inst(P)$.

Otherwise, we use the completeness hypothesis on S to show that $inst(P)$ contains $\nu_{|R}(C)$. We also use Lemma 9.10 to show that, using $(T_{inst(P)} \uparrow k)(I)$ as an interpretation, this partially instantiated clause matches $\nu_{|\overline{R}}$ to produce f. □

Theorem 9.12. (Transformation soundness) For any program P, initial interpretation I and number of steps k, any fact deduced by the transformed program in k iterations of T_P was already computed after the same number of steps in the original program, i.e. $(T_{inst(P)} \uparrow k)(I) \subseteq (T_P \uparrow k)(I)$.

The proof of this lemma works in a both similar and dual way w.r.t. that of Theorem 9.11, as we get two substitutions (one for the clause instantiation, one matching the transformed clause) that need to be combined to retrieve the substitution used in the original program. The combination operator between two functions to be used is the following.

Definition 9.13. (Union of substitutions) Given two substitutions σ_1 and σ_2, the union of σ_1 and σ_2, written $\sigma_1 \cup \sigma_2$, is defined as

$$\sigma_1 \cup \sigma_2\ (x) = \begin{cases} \sigma_1(x) & \text{if } \ \sigma_1(x) \text{ is a constant} \\ \sigma_2(x) & \text{otherwise} \end{cases}$$

Remark 9.14. Despite the first case of the definition, the union of two substitutions returns a term, i.e. a constant *or* a variable. A mapping to a constant will be prioritized over one to a variable, but if $\sigma_1(x)$ and $\sigma_2(x)$ are both variables, then so will $\sigma_1 \cup \sigma_2\ (x)$.

Remark 9.15. If $\sigma_1(x)$ and $\sigma_2(x)$ are both variables or both constants, this operation defines different priorities between the two given substitutions. More concretely, if σ_1 and σ_2 both map x to different constants c_1 and c_2, then $\sigma_1 \cup \sigma_2$ will map x to c_1, whereas if they map x to variables v_1 and v_2, the latter will be returned by the union.

In that sense, calling it "union" might be slightly misleading, as it is not a commutative function. However, in practice, the domains of the two substitutions are disjoint wherever we use this notion in our work, meaning that defining the priority either way has no impact.

Proof. The soundness of the transformation is proved using an induction on the number of steps k. In the base case, the definitions again imply that $(T_P \uparrow 0)(I) = (T_{inst(P)} \uparrow 0)(I) = I$, which is more general than our goal.

In the recursive case, let f be a fact in $(T_{inst(P)} \uparrow k+1)(I)$. Then, f was either already in the previous iteration $(T_{inst(P)} \uparrow k)(I)$, or it has just been deduced and added. In the first scenario, the induction hypothesis gives us that f is also in $(T_P \uparrow k)(I)$, and thus in $(T_P \uparrow k+1)(I)$ by definition.

In the second scenario, we can extract a clause C from $inst(P)$ and a substitution ν such

that ν matches C w.r.t. $(T_{inst(P)} \uparrow k)(I)$ and f is the head of $\nu(C)$. Clause C is the result of applying $inst$ to a clause C_o of the original program, i.e. $C = inst(C_o)$. Just like in the definition of $inst$, we need to consider whether C_o contains variables to instantiate.

If C_o has no relevant variable, i.e. $|vars(C_o) \cap R| = 0$, $C = C_o$. We then deduce f in P reusing C, ν. The atoms in the body of the instantiated clause being in $(T_P \uparrow k)(I)$ (induction hypothesis), we can indeed do so.

Otherwise, there is a substitution σ such that $C = \sigma(C_o)$, with $dom(\sigma) = R \cap vars(body(C_o))$. To deduce f in P w.r.t. $(T_P \uparrow k)(I)$, we use the C_o clause with the $\nu \cup \sigma$ substitution. Since ν and σ have disjoint domains, we can easily show that $\nu \cup \sigma(C_o) = \nu(\sigma(C_o)) = \nu(C)$. We then obtain the same instantiated clause as in the transformed program $inst(P)$. Thanks to the induction hypothesis, $(T_{inst} \uparrow k)(I) \subseteq (T_P \uparrow k)(I)$, the intermediate interpretation for the original program contains at least the facts that were used for the deduction in the transformed program. We use them, with the aforementioned instantiated clause $\nu \cup \sigma(C_o)$, to deduce f. □

Now that the partial program instantiation is defined and justified on paper, we can formalize and certify it within Coq and MathComp.

9.3 Coq implementation and certification

We first assume the variables introduced earlier: a safe (in the sense defined in Section 2.1.3, cf. Figure 3.10) program, an initial interpretation, a default constant (used to transform substitutions into groundings, cf. Figure 3.16), the set of variables to instantiate `Rv` (previously R) and the provided substitutions `subs` (previously S). Figure 9.6 sums up these assumptions.

```
Variable p : program.
Hypothesis psafe : prog_safe p.

Variable i : interp.
Variable def : syntax.constant.

Variable Rv : {set 'I_n}.

Variable subs : {set sub}.
```

Figure 9.6: Coq hypotheses the partial program instantiation

Surprisingly, the original development of DatalogCert does not provide a boolean substitution matching function. The closest thing is `gcl_true` (which is lifted to `cl_true` and `prog_true`, see Figure 3.17), used to implement the model-theoretic semantics. However, it only relies on groundings rather than (partial) substitutions, and is not explicitely related (i.e. *via* lemmas) to the other components of the formalization.

This led us to focus on the constructive matching mechanism, using `match_pbody` (cf. Figure 3.23) to define the substitutions involved in the proofs. However, proving results stated with this function was much harder than expected. Even trying to prove a simple, technical result such as Lemma 9.10 with a classical induction on the atom list required us to find non trivial

formulations and abstractions. This is because `match_pbody` collects contraints as it goes through the atom list, meaning that we have to relate the constraints collected on both sides of the implication.

Using this constructive matching, we were able to (painstakingly) prove most of the technical lemmas needed for the adequacy proofs, but not all of them. Even if we had managed to do so, the effort required felt uncorrelated to the intrinsic difficulty of the targeted results. We eventually completed the adequacy proof after introducing a boolean matching predicate.

This predicate, shown in Figure 9.7, first checks that the variables of the given clause all appear in the domain of the given substitution. If it is the case, the substitution can safely be made into a grounding using function `to_gr` and the `def` constant, as it will not be used in the instantiation of the clause. Function `gr_tl` simply maps a given grounding to a sequence of atoms, here the body of the clause to instantiate.

```
Definition bmatch i cl s : bool :=
   (cl_vars cl \subset dom s)
&& all (mem i) (gr_tl (to_gr def s) (body_cl cl))
```

Figure 9.7: Coq version of the boolean matching

Figure 9.8 shows how this boolean matching was related to its constructive counterpart, and thus to the fixpoint semantics. The first lemma simply states that any built substitution indeed matches the given clause. The second lemma, given a boolean match, ensures that the constructive match would succeed on the given clause w.r.t. the interpretation, and that the result would be a *subsubstitution*, i.e. be sound w.r.t. the first one (cf. Figure 3.15).

```
Lemma match_body_bmatch def (cl : clause) i s :
  s \in match_body i (body_cl cl) -> bmatch def i cl s.

Lemma bmatch_match_body def (cl : clause) i s:
  bmatch def i cl s -> exists2 r : sub, r \in match_body i (body_cl cl) & r \sub s.
```

Figure 9.8: Relating the two kinds of matching

To implement the completeness hypothesis over the provided set of substitutions, we also need the full semantics of the program. We adapt the proof of `incr_fwd_chain_complete` (cf. Section 3.3.2.2) to define it as $|\mathbb{B}_P|$ iterations of T_P and characterize its maximality w.r.t. `match_body`, as shown in Figure 9.9.

```
Definition sem (m : nat) := iter m (fwd_chain def p) i.

Definition ffp := sem #|bp|.

Lemma nomega_fp :
 forall (cl : clause) (m : nat), cl \in p
-> match_body (sem p def m i) (body_cl cl) \subset match_body ffp (body_cl cl).
```

Figure 9.9: Definition and absorption of a program's semantics

Finally, we need the restriction of a substitution to a given set of variables (cf. Definition 9.3). Figure 9.10 shows its implementation with `finfun` notations.

```
Definition sub_filter (s : sub) (t : {set 'I_n}) :=
  [ffun v => if v \in t then s v else None].
```

Figure 9.10: Restricting a substitution in Coq

With all these definitions in hand, the hypothesis on `subs` can be defined as shown in Figure 9.11.

```
Hypothesis subs_comp :
  [forall cl in p,
    (#|tail_vars (body_cl cl) :&: Rv| > 0) ==>
      [forall s : sub, (bmatch def ffp cl s) ==> (sub_filter s Rv \in subs)]].
```

Figure 9.11: Coq implementation of the completeness hypothesis on the overapproximation

The implementation of the actual instantiation, shown in Figure 9.12, is straightforward as well, thanks to MathComp's set comprehension features.

```
(* [stv] returns the to-be-replaced variables of a clause *)
Definition stv cl := Rv :&: tail_vars (body_cl cl).

Definition inst_rule (cl : clause): seq clause :=
  (* s is only about interesting variables *)
  if (#|tail_vars (body_cl cl) :&: Rv| > 0) then
    [seq scl s cl | s <- enum subs & dom s == stv cl]
  else [:: cl].

(* [tprog] is obtained by instantiating all the rules of [p] *)
Definition tprog : program :=
  flatten [seq (inst_rule cl) | cl <- p].
```

Figure 9.12: Coq implementation of the partial program instantiation

Figure 9.13 shows the Coq definition of Theorems 9.11 and 9.12, whose proofs are fundamentally similar to the paper versions above. The reasoning is actually quite recognizable when reading the proof script, which is always very satisfactory when working with Coq. The main difficulty was the use of substitutions and matchings, which we already discussed.

```
Lemma ccompleteness (m : nat) : (sem p def m i) \subset (sem tprog def m i).
Lemma csoundness (m : nat) : (sem tprog def m i) \subset (sem p def m i).

Theorem cadequacy (m : nat) : (sem tprog def m i) = (sem p def m i).
```

Figure 9.13: Coq definition of the adequacy of the partial program instantiation

This rewriting requires – both on the paper and Coq levels – an overapproximation of the transformed program's behavior. The next chapter introduces, formalizes and discusses a static analysis that provides one.

Chapter 10

Static analysis

> - Peux-tu expliquer correctement la différence entre le présent et le passé du conditionnel en anglais ? me demanda-t-elle soudain.
>
> - Je crois que oui, lui répondis-je.
>
> - Je voudrais bien savoir à quoi cela te sert dans ta vie quotidienne.
>
> - C'est vrai que cela ne m'est pas très utile, mais je crois que, plutôt que d'en chercher l'utilité concrète, il faut le considérer comme un exercice destiné à vous faire appréhender les choses avec méthode.
>
> Haruki Murakami, *La ballade de l'impossible*,
> traduit du japonais par Rose-Marie Makuno-Fayolle

In this chapter, we introduce a new static analysis for Datalog. This analysis approximates the behavior of a variable in a program, i.e. the set of values it will be instantiated with during the program's execution, by simulating a more local and less constrained version of the fixpoint semantics for Datalog introduced in Section 2.2.2.

Section 10.1 first introduces some (harmless) assumptions made by the analysis about the studied program. Then, Section 10.2 outlines *via* an example the main ideas behind the analysis. These ideas are formalized in Coq in Section 10.3, and certified in the same framework in Section 10.4. This work led us to design another, stronger version of the analysis, whose introduction and discussion are relagated to Chapter 12.

10.1 Hypotheses and notations

The first hypothesis slightly simplifies the analysis by restricting the syntax of Datalog, but not its expressivity.

Hypothesis 10.1. (Constant-free heads) We first assume that the heads of rules in the program do not contain constants, but only variables.

Lemma 10.2. Hypothesis 10.1 is harmless, i.e. any Datalog program can be rewritten into

an equivalent program that enforces it.

Proof. The proof of Theorem 12.5.2 of [Abiteboul et al., 1995] provides a transformation that eliminates any given constant from a program. Roughly, the idea of this transformation is to introduce an extensional predicate C_a for each constant a that appears in a head of the program, and transform any rule of the form

$$H(..., a, ...) \leftarrow B_1(...), ..., B_n(...)$$

into

$$H(..., x, ...) \leftarrow B_1(...), ..., B_n(...), C_a(x)$$

where x is a fresh variable. The only fact about C_a found in the EDB is $C_a(a)$, meaning that x will be constrained to be instantiated to a during the program's execution. □

Our other hypothesis on the analyzed programs is even simpler but also helps keeping the formalization of the analysis minimal.

Hypothesis 10.3. (Unicity of variables across rules) We only consider programs where individual variables may only appear in a single rule.

Lemma 10.4. Hypothesis 10.3 is a harmless assumption, i.e. any Datalog program can be rewritten into an equivalent program where no variable is shared across multiple rules.

Proof. The property can be enforced by indexing the variable names w.r.t. the rules. For example, Figure 10.1 shows how an arbitrary program (Figure 10.1a) is transformed (Figure 10.1b) so that a variable does not appear in two different rules. □

$$\begin{array}{l} P(X,Y) \leftarrow Q(X,Y)\,. \\ Q(X,Y) \leftarrow Q(Y,X)\,. \\ Q(X,Y) \leftarrow f(X,X,Y)\,. \end{array}$$

(a) A program with shared variables...

$$\begin{array}{l} P(X_1,Y_1) \leftarrow Q(X_1,Y_1)\,. \\ Q(X_2,Y_2) \leftarrow Q(Y_2,X_2)\,. \\ Q(X_3,Y_3) \leftarrow f(X_3,X_3,Y_3)\,. \end{array}$$

(b) ...and with unicity across rules

Figure 10.1: Indexing variable names

Finally, we introduce two notations that will be used throughout the rest of this document.

Notation 10.5. (Predicate index) Given a predicate P and an integer i, we denote as $P.i$ the i^{th} index, or argument position (starting at 0) of P.

Notation 10.6. (*tocc* – Term body occurrences) We denote as *tocc*s (for *term occurrences*) the occurrences of terms within the bodies of rules. These occurrences are 3-tuples in $\mathbb{N}^3$, where the components are the indexes of, respectively, the rule, the atom (within the body of the rule) and the argument (within the atom), starting at 0.

Example 10.7. In the program of Figure 10.1a, the *tocc*s for the occurrences of X within the f atom are $\langle 2, 0, 0\rangle$ and $\langle 2, 0, 1\rangle$.

10.2 Intuition and Example

We first introduce, *via* an example, the general principle of the static analysis. We then focus on a specific aspect, i.e. the way it deals with Datalog recursion. Finally, we outline how we extract a set of substitutions from the analysis results.

10.2.1 General principle

The work on this static analysis started by noticing that the T_P operator can be split into three distinct constraints, and realizing that implementing a subset of these constraints could lead to an overapproximating but much faster analysis. We first use the program of Figure 10.2, where s, q and r are intensional predicates, and f_1 to f_3 extensional, to illustrate the three properties T_P must fullfil. We will then give an intuition of the static analysis itself, and finally provide a detailed example of its application, using once again the program of Figure 10.2.

1. Given any EDB, the set of values with which X_1 can be instantiated in practice is a subset of the intersection of the sets of values that go through its instances within the body of the rule, i.e. the values of *tocc*s $q.0$ and $f_1.0$.

2. Focusing on the first, the set of values with which $q.0$ can be instantiated is a subset of the union of the values *returned* by the second and third rules.

3. Given a value for X_1 and another for Y_1, they must be compatible, i.e. form a valid tuple of arguments for f_1.

$$
\begin{array}{ll}
s(X_1, Y_1) & \leftarrow q(X_1),\ \ f_1(X_1, Y_1)\,. \\
q(X_2) & \leftarrow r(X_2, X_2)\,. \\
q(X_3) & \leftarrow f_2(Y_3, Y_3, Z_3),\ \ f_3(Y_3, X_3, Z_3)\,. \\
r(X_4, Y_4) & \leftarrow f_2(X_4, Y_4, X_4),\ \ f_3(Z_4, X_4, Z_4)
\end{array}
$$

Figure 10.2: Defining s(X,Y)

Unlike the third constraint, the first two deal with variables and *tocc*s individually, meaning that they involve much less computation than the entirety of T_P, while hopefully still providing interesting constraints. Our idea is then to design an analysis that fulfills the first two constraints.

Since these two constraints fundamentally are intersections and unions of value sets (or, from a logical standpoint, conjunction and disjunction), the idea is to build, for any chosen variable, a tree with nodes labeled with $\cap$ and $\cup$. The leaves, which correspond to the starting points of any series of deduction, represent columns in the EDB tables, e.g. $f_1.0$ or $f_2.1$. This way, the tree represents the way values flow from the EDB to a variable during the execution of the program, without enforcing the unification across different variables.

The branches are annotated with the index of the corresponding clause (for the descendants of $\cup$-nodes) or atom (descendants of $\cap$-nodes). Although this helps understanding and relating the tree with the analyzed program, the main point of these annotations is that they can be used to simulate a weaker version of the third constraint of T_P on top of the actual analysis, as we will discuss in Section 12.2.

Example 10.8. The *flow* of X_1 in the program of Figure 10.2 is shown in Figure 10.3. Let us relate it to the actual program, using the annotations.

Variable X_1 has two occurrences in the body of the first rule. The set of possible values for X_1 is a subset of the intersection of those for the two occurrences, meaning that the root of the tree has to be labeled with $\cap$. The first occurrence is in the q atom, which itself is the first element of the body, and the second occurrence is in the f_1 atom, second element of the body. This means that the node must have two children, annoted 0 (for the occurrence in q) and 1 (occurrence in f_1). Since f_1 is an extensional predicate, the second child is a leaf annotated with the corresponding EDB table. Let us now focus on the left child.

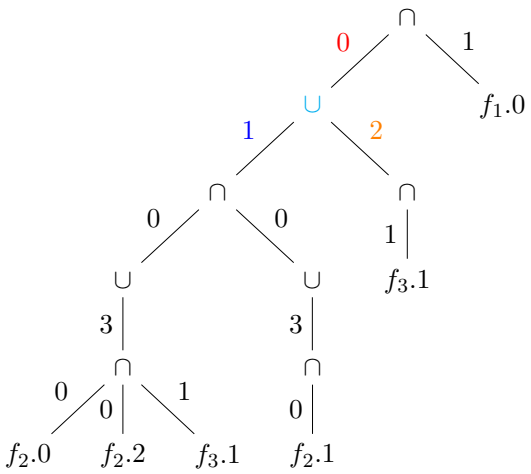


Figure 10.3: Analysis of a variable

Since the first occurrence of X_1 was as the first (and only) argument of an atom of an intensional predicate (namely q) the subtree should represent the different ways to obtain values for $q.0$. This implies that the root will be annotated $\cup$ (in general, there will be a strict alternation between intersections and disjunctions throughout the analysis), and will have one child for each rule defining q.

We then have two descendants, annotated 1 and 2, which refer to the second and thrid rules of the program, respectively. We first focus on the latter. This rule defines $q(X_3)$ using two atoms. However, only the second one, defined with the extensional predicate f_3, contains an occurrence of X_3, at its second argument. We then return a $\cap$ node, which only leads to a leaf that refers to the $f_3.1$ column of the EDB.

As for the left child, annotated with 1, it refers to a rule that contains only one atom in its body, but two occurrences of the relevant variable, X_2. We then have two descendants, both annotated 0, which compute overapproximations of the values for $r.0$ and $r.1$. Since the only rule defining r is the fourth one, the roots of both subtrees *start* are a $\cup$-node and have only one successor, labeled with 3.

The left subtree is an analysis of $r.0$, and thus of the first argument of the head of the last rule, X_4. This variable has three occurrences in the body, meaning that there are just as many descendants. On the other hand, the second argument, Y_4, has only one occurrence.

10.2.2 Dealing with recursion

The general principle introduced in Example 10.8 will obviously not fare well (i.e. loop indefinitely) in the presence of recursion, which happens to be a core feature of Datalog. To circumvent this issue, the analysis stores all previously visited *tocc*s when recursively calling itself, to avoid having a program location twice in a branch of the returned tree. This bounds the derivations despite potentially recursive programs.

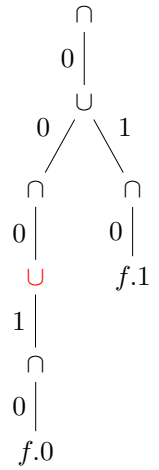
The rationale behind this fix is that, to find an approximation of the values going through some *tocc*, one should not look at the recursive part of the corresponding predicate, but rather the other predicates that "ground" it. This idea is formalized is Section 10.4, but let us provide a first intuition.

Example 10.9. Figure 10.4b shows the analysis of variable X_1 in the program of Figure 10.4a, where f is an extensional predicate. The analysis starts with the p atom in the body of the first clause. It then considers two different ways to deduce a p fact, i.e. the two clauses. When looking at the first clause, the analysis moves on to the variable matching $p.1$, i.e. Y_1. It also occurs in the p atom, so we could expect to have again two children, one for each rule.

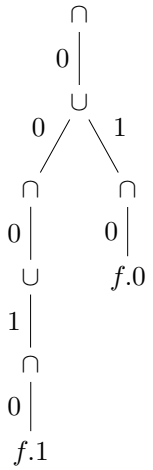
However, going through the first clause once again would require analyzing $p.0$. Since this is precisely what we are doing, all occurrences of X_1 have already been visited and stored. We then drop this possibility, meaning that the ∪-node we were considering only has one child. The analysis eventually captures the fact that values of p can be permuted, and that the set of values of X_1 is a subset those of $q.0$ and $q.1$.

$$p(X_1, Y_1) \leftarrow p(Y_1, X_1)\,.$$
$$p(X_2, Y_2) \leftarrow f(X_2, Y_2)\,.$$

(a) Program P_{rec}



(b) Analysis of X_1



(c) Analysis of Y_1

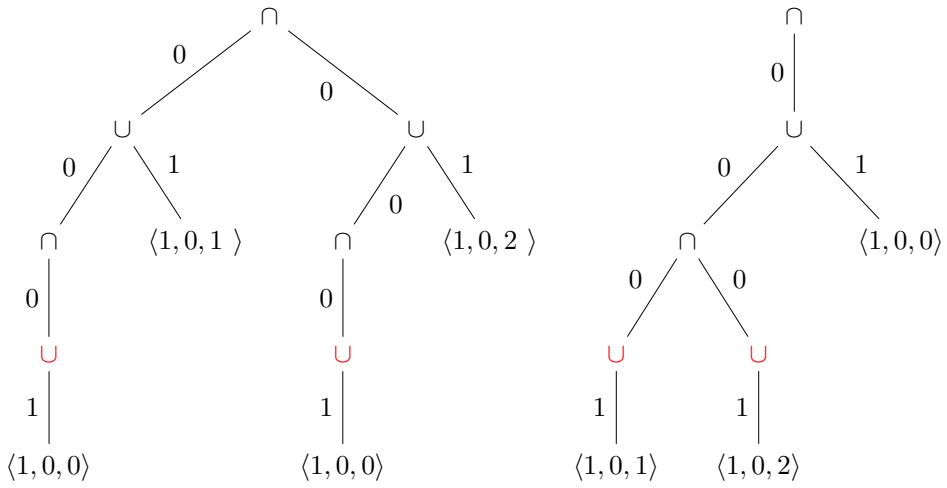
Figure 10.4: Analysis of a recursive program

Example 10.10. Let us consider the program of Figure 10.5.

$$p(X_1, Y_1, Y_1) \leftarrow p(Y_1, X_1, X_1)\,.$$
$$p(X_2, Y_2, Z_2) \leftarrow q(X_2, Y_2, Z_2)\,.$$

Figure 10.5: Shifting variables

The first rule contains two occurrences of variable Y_1 in its head, and two occurrences of X_1 in its body, not at the same indexes. The static analysis does handle this program, and returns the analyses shown in Figure 10.6. In both trees, the nodes in red are those where a *cut* happens to avoid looping.



(a) Analysis of X_1 **(b)** Analysis of Y_1

Figure 10.6: Another program analysis

Replacing the occurrences by the predicate and index they correspond to, the tree of Figure 10.6a represents

$$(f.0 \cup f.1) \cap (f.0 \cup f.2) = f.0 \cup (f.1 \cap f.2)$$

The other tree corresponds to the same formula, and, in both cases, the result corresponds to the actual behavior of the analyzed variable.

Now that the actual analysis of a variable has been fully introduced, we can discuss how the returned trees are used to extract a set of substitutions, as required by the program transformation presented in Chapter 9.

10.2.3 From trees to values

Given a tree representing the flow of a variable, we can easily extract a set of values with a structural induction. Intuitively, the $\cap$ and $\cup$ nodes are treated as set intersection and union, respectively. The leaves return the values in the corresponding column of the EDB tables, i.e. the extraction of a $f.i$ leaf will return the set of constants appearing at the i^{th} position of a f fact in the EDB.

Given the analyses of multiple variables appearing in the same rule, partial substitutions can be built as the cartesian product of the value sets extracted from the different trees. This simplistic procedure is not entirely satisfactory, as it may produce many irrelevant value tuples. Section 12.2 discusses how a more complex extraction process can use the annotations of multiple trees to circumvent this issue.

10.3 Formalization

There is usually a gap in clarity and simplicity between a paper definition and its concrete implementation. However, thanks to MathComp's set notations, both versions are actually very similar. We then reckon that presenting both would make for a rather redundant read, and restrict ourselves to the latter, i.e. the Coq/MathComp formalization.

10.3.1 From trees to DNF

Replacing $\cup$ and $\cap$ with $\vee$ and $\wedge$, the trees produced by the analysis can be seen as propositional formulae, where the EDB columns are atomic elements. With that in mind, the Coq formalization works with Disjunctive Normal Forms (DNF). In this form, the relation with the actual computation seems less natural, as we lose the alternation between intersections and disjunctions, but is surprisingly much easier to prove.

Intuitively, the intersections in the DNF encode the different *paths* values can take, i.e. which rule is used to deduce an element of a predicate that is defined by multiple clauses. This way, one of the core lemmas of the certification process will state that every such possibility is indeed captured by the returned DNF.

Building a DNF requires the computation of every combination of propositional formulae across different disjunctions – here, sets. MathComp already provides a cartesian product between two sets, which is shown in Figure 10.7.

```
Definition setX := [set u | u.1 \in A1 & u.2 \in A2].
```

Figure 10.7: Cartesian product between two sets in MathComp

Figure 10.8 shows our generalization to any finite number of sets. It takes a sequence of sets, denoted as `ss`, and returns a set of tuples, the size of which is the length of `ss`. Each tuple contains an element of each set, these elements being ordered the same way as their original sets within `ss`. Using tuples on a `finType` rather than sequence types allows having them in a set (cf. Example 3.6).

Notation 10.11. `tnth x j`, where `x` is a `n.-tuple` and `j` is a `'I_n`, returns the j^{th} element of `x`, and `in_tuple s` returns sequence `s` seen as a `(size s).-tuple`.

```
Definition gen_setX {A : finType} (ss : seq {set A})
 : {set (size ss).-tuple A} :=
  let m := size ss in
  [set x : m.-tuple A | [forall j : 'I_m, tnth x j \in tnth (in_tuple ss) j]].
```

Figure 10.8: Cartesian product between an arbitrary number of sets

Remark 10.12. Figure 10.8 leverages `finType`s to use set notations, but the resulting definition, like `setX`, is not very constructive. In fact, the cartesian product is defined here using its specification. Thanksfully, such universal and useful constructs are easily found in traditional programming languages.

10.3.2 Handling occurrences

Implementing the static analysis requires a precise definition of *tocc*s, i.e. program occurrences, as well as some related technical results. We present the main definitions, both as for exhaustivity, and to illustrate how, in the spirit of DatalogCert, the focus on `finType`s sometimes implies a heavy, potentially cumbersome use of dependent types to bound the manipulated objects.

First, we need to define and bound the occurrence type. As shown in Figure 10.9, the coordinates are encoded as ordinals. The rule index is simply bounded by the number of rules. The index within the body of the rule uses the `bn` parameter introduced in Section 7.3. Finally, the index of the term within the arguments of an atom is bounded by the maximal arity within all the available predicates, `max_ar`, which was already defined in DatalogCert.

```
Notation max_ar := (\max_(s in symtype) arity s).

Record t_occ p :=
  T_occ {r_ind : 'I_(size p) ; b_ind : 'I_bn ; t_ind : 'I_max_ar}.

Definition tocs p := {set (t_occ p)}.
```

Figure 10.9: Bounding program occurrences

The `t_occ` type was easily shown finite using `CanFinMixin`. This finiteness results then allows the definition of sets of occurrences, called `tocs`.

We can now easily define functions that return the atom or predicate corresponding to a given program occurrence. As seen in Figure 10.10, we use the `nth_error` function, which does not require a default element like `nth` but returns an option type[1]. Functions `at(om)_at` and `p(redicate)_at` then do as well.

```
Definition at_at (o : t_occ) : option atom :=
  match nth_error p (r_ind o) with
    | None => None
    | Some cl => nth_error (body_cl cl) (b_ind o) end.

Definition p_at (t : t_occ) :=
  match (at_at t) with
    | None => None
    | Some ato => Some (sym_atom ato) end.
```

Figure 10.10: Computing an occurrence's predicate

Remark 10.13. The use of an option return type generates some frustration during the proofs, as in practice we make sure to use these fonctions only with occurrences that actually correspond to something in the program, and have to care about irrelevant cases in matches and such. However, these functions being technically partial, there was probably no way to circumvent this – using a default return value rather than an encapsulation within an option type would have led to the same problem.

[1] https://coq.inria.fr/library/Coq.Lists.List.html#nth_error

We also need functions that collect the set of occurrences of a given variable within the program. However, due to the bounding of occurrences as seen above, this process requires a surprising quantity of machinery. In particular, after trying, we will argue that writing this function *ex nihilo* seems illusory. We then build up to it, starting with the minimal relevant structures, i.e. lists of terms of atoms, as seen in Figure 10.11. Within these structures, the occurrences only contain an index, bounded by the maximal predicate arity.

```
Definition occsInTermList (v : 'I_n) (l : seq term) : {set 'I_max_ar} :=
  [set i : 'I_max_ar | nth_error l i == Some (Var v)].

Definition occsInAtom (a : atom) (v : 'I_n) : {set 'I_max_ar} :=
  @occsInTermList v (arg_atom a).

Lemma occsInAtomV occ a v :
   occ \in occsInAtom a v -> nth_error (arg_atom a) occ = Some (Var v).
```

Figure 10.11: Finding occurrences in an atom

We then lift this to lists of atoms. To do so, we add a second ordinal (in the sense of MathComp, see Example 3.4) to the occurrences, corresponding to the index of the atom containing the variable occurrence. When exploring a new atom, this (atom) index is set to 0, and those of the previously seen atoms are incremented.

Figure 10.12 first shows how we define the increment of an Ordinal. Lemma `ord_shift1` *shifts* a bound on `x`, `ord_shift` uses it to build the actual ordinal, and `shift1` simply lifts it to sets of pairs.

```
Lemma ord_shiftl {i : nat} : forall x : 'I_i, x.+1 < i.+1.

Definition ord_shift {i : nat} (x : 'I_i) : 'I_i.+1 :=
  Ordinal (ord_shiftl x).

Definition shift1 {k : nat} {A : finType} (l : {set ('I_k * A)%type})
  : {set ('I_k.+1 * A)%type} := [set ((ord_shift x.1), x.2) | x in l].
```

Figure 10.12: Incrementing an ordinal

This function can then be used in the actual collection of occurrences of a variable in a sequence of atoms, shown in Figure 10.13. For each new atom `a`, we compute the set of occurrences it contains and *assign* it `0` seen as an ordinal (`Ordinal (ltn0Sn _)`), whereas the previously computed indexes are *shifted* by one on the left component.

```
Fixpoint occsInAtomList (al : seq atom) (v : 'I_n)
 : {set ('I_(size al) * 'I_max_ar)} :=
  match al with
  | [::] => set0
  | a::al => ([set (Ordinal (ltn0Sn _), x) | x in (occsInAtom a v)]
         :|: (shift1 (@occsInAtomList al v))) end.
```

Figure 10.13: Finding occurrences in an atom list

Figure 10.14. shows the certification of `occsInAtomList`, w.r.t. `t_at`. We use Lemma `wlist_to_seq_size`, which takes an element `l` of type `Wlist A m` and returns `size l < m`. Then, `wlist_to_seq_size (body_cl cl)` states that the size of the body of `cl` is less than `bn`, which `widen_ord` uses to *widen* the body index from `'I_(size (body_cl cl))` to `'I_bn`.

```
Lemma occsInAtomListV {p} (cl : clause) (cln : 'I_(size p)) v aton termn :
   Some cl = nth_error p cln
-> (aton, termn) \in occsInAtomList (body_cl cl) v
-> @t_at p {| r_ind := cln;
              b_ind := widen_ord (wlist_to_seq_size (body_cl cl)) aton;
              t_ind := termn |} = Some (Var v).
```

Figure 10.14: Certification of occsInAtomList

Function `occsInRule`, shown in Figure 10.15, simply lifts the previous definition from lists of atoms to a rule. The size of the first component of the returned occurrences then has to be harmonized, meaning that the *ordinal widening* is also used here to extend all computed body bounds to `bn`. The associated lemma is also similar to `occsInAtomListV`, although more straightforward.

```
Definition occsInRule (v : 'I_n) (cl : clause) : {set ('I_bn * 'I_max_ar)} :=
  [set ((widen_ord (wlist_to_seq_size (body_cl cl)) (fst x)), (snd x))
                              | x in (occsInAtomList (body_cl cl) v)].

Lemma occsInRuleV {p} cl v aton termn :
    ((aton, termn) \in occsInRule v (tnth (in_tuple p) cl))
 -> t_at {| r_ind := cl;
            b_ind := aton;
            t_ind := termn |} = Some (Var v).
```

Figure 10.15: Finding occurrences in a rule

Finally, Figure 10.16 lifts these definitions to full programs. To do so, `occsInRule` is applied to every rule of the program, and the clause index is added to the returned set of coordinates. Lemma `occsInProgramV` is the final, straightforward specification of the set of functions introduced in this Section.

```
Definition attach_cl_nb p (cl_nb : 'I_(size p))
                          (occs : {set ('I_bn * 'I_max_ar)}): tocs p :=
  [set @T_occ p cl_nb (fst x) (snd x) | x in occs].

Definition occsInProgram p (v : 'I_n) : tocs p :=
  \bigcup_(cln in 'I_(size p))
     attach_cl_nb cln (@occsInRule v (tnth (in_tuple p) cln)).

Lemma occsInProgramV {p} v (xocc : t_occ p) :
   xocc \in occsInProgram p v
-> t_at (xocc) = Some (Var v).
```

Figure 10.16: Finding occurrences in a program

Remark 10.14. The machinery used to return finite types is specific to Coq, and would not appear in a natural implementation. Although it may hurt the performances of the extracted program, especially considering how heavily these functions are used, they are so fundamentally simple and easy to implement that one may safely consider recoding them in an actual programming language. We had to implement them in Coq for the analysis, but the verification of these definitions is not an earth-shattering contribution.

Finally, we will need the functions shown in Figure 10.17. Given default variable and terms, `term_to_var cl j` returns the j^{th} term in the head of clause `cl`. Since Hypothesis 10.1 states that there are no constants in the heads of rules, we can add a safe *cast* to variables for typing purposes. In particular, the `dv` and `dt` default variable and term assumptions are only introduced for the exhaustiveness of pattern matching and use of the `nth` function.

With these tools in hand, we can move on to the definition and verification of the actual program analysis.

```
Variable dv : 'I_n.
Variable dt : term.

Definition term_to_var (t : term) :=
  match t with | Val c => dv | Var v => v end.

Definition get_cl_var (cl : clause) (j : nat) : 'I_n :=
  term_to_var (nth dt (arg_atom (head_cl cl)) j).
```

Figure 10.17: Fetching head variables and body predicates

10.3.3 Analysis implementation

As stated in Section 10.3.1, an analysis result comes in the form of a Disjunctive Normal Form. Although there are infinitely many DNF formulae, the leaf type (pairs with a predicate symbol and an ordinal bound by the maximal predicate arity) being a `finType` and the absence of repetition across branches in the trees produced by the analysis allow us to encode the formulae in a finite structure, namely a set of sets.

In that setting, the *main*, or *outer* set represents the disjunction, implying that the *inner* sets are the multiple intersections. These *intersections* contain the full branches as `uniq_seq`s (cf. Section 7.1.2) rather than simply the leaves, as it will help us relate the structure to the analyzed program's behavior. As previously intuited, using DNFs is akin to having an unfolding of the different possible behaviors of the studied program, which simplifies the core certification lemmas.

Figure 10.18 shows the Coq definition of the static analysis, `analyze_var`, which is broken down in the following paragraphs.

Arguments The actual implementation is (obviously) found in the `analyze_var_prev` function, which is parametrized by the set of previously visited *tocc*s `prev`. Along with `prev` and the studied variable `v`, the function uses an argument `count`, which is used to ensure termination in a simple way. The `analyze_var_prev` function first checks whether it has reached 0 and, if it is the case, returns the empty set. In Section 10.4, when discussing the

verification of the analysis, we will show how `count` can be instantiated by a program-specific bound while retaining compleness.

```
Fixpoint analyze_var_prev (prev : {set tocc p}) (v : var)
                          (count : nat) : {set {set (uniq_seq tocc)}} :=

  (* Ensuring termination *)
  match count with | 0 => set0 | count.+1 =>

    (* Computing non-visited occurrences of v *)
    let occs := occsInProgram p v :\: prev in

    (* Analysis of a predicate and index pair *)
    let analyze_pi (prev : {set tocc}) (o : tocc) :=
        match p_at o with
         | None => set0
         | Some f =>
            match predtype f with
             | Edb => [set [set unil]]
             | Idb => let arec :=
                        [set (analyze_var_prev prev (get_cl_var cl (t_ind o))) count
                               | cl in p & head_predicate cl == f]
                      in \bigcup_(x in arec) x end end in

    (* Adding the current tocc on top of a DNF *)
    let all_add_o (dt : {set {set (uniq_seq tocc)}}) (o : tocc)
                  {set {set (uniq_seq tocc)}} :=
      [set [set o::br | br in ct] | ct in dt] in

    (* Recursive call *)
    let arec := [seq all_add_o (analyze_pi (occ |: prev) occ) occ
                                                | occ <- enum occs] in
    bigcup_cart (gen_setX arec) end.

(* Version used in practice, with prev set to empty set for maximal precision *)
Definition analyze_var (v : var) (count : nat) : {set {set (uniq_seq tocc)}} :=
  analyze_var_prev set0 v count.
```

Figure 10.18: The static analysis in Coq

Fetching occurrences Once it has checked that `count` has not reached 0, the function computes the set of occurrences of the `v` variable that are not in the `prev` argument, i.e. occurrences that have not been previously visited (`:\:` is the MathComp notation for set difference). Then, for each occurrence, we need to both make recursive calls over the corresponding predicate and index par, and store the current *tocc* on top of the result. Finally, we will have to combine the results over all occurrences.

Recursive calls Function `analyze_pi` focuses on a single *tocc*, `o`. It then computes the predicate corresponding to the occurrence, written `f` (due to the definition of `occsInProgram`,

the `None` case is never used in practice, and is only here for the matching's exhaustivity). We then take a look at the kind of predicate, i.e. intensional or extensional, that `f` is.

If it is extensional, then we actually do not want to perform a recursive call, and just return a leaf. Since the addition of the currently studied occurrence is performed later in the code (in the case where it has to be put on top of multiple branches), we simply return an empty `uniq_seq` (cf. Figure 7.12) encapsulated in two layers of sets.

If `f` is intensional, we go over the program, looking for clauses whose head is an occurrence of the predicate. For each of these clauses, we compute the variable corresponding to the currently studied occurrence `o`, i.e. the variable in the head whose index is the same as the argument index (`t_ind`) of `o`. We then perform the analysis on these variables, using a `prev` argument enriched with the current occurrence (cf. the call of `analyze_pi`). The results, which are all sets of sets representing DNFs, must be merged. Since the different rules defining a predicate correspond to multiple ways to deduce a fact, we want a disjunction over all these possibilities. We then unify all the disjunctions into a single one, by merging the *outer sets* using a `bigcup` notation.

Storing the current occurrence Now that we have a flow leading to the currently studied occurrence, we need to put it on the top of the different branches. This is done, as seen with function `all_add_o`, with a simple set notation.

```
Definition bigcup_tup {m} {A : finType} (t : m.-tuple {set A}) : {set A} :=
  \bigcup_(x <- tval t) x.

Definition bigcup_cart {m} {A : finType}
                       (s : {set m.-tuple {set A}}) : {set {set A}} :=
  [set bigcup_tup y | y : m.-tuple {set A} in s].
```

Figure 10.19: From cartesian product to set of sets

Merging analyses across occurrences We now have a DNF, seen as a set of sets, for each occurrence of the studied variable `v`. All these occurrences are from the same rule (thanks to the assumption that a variable does not appear in two different clauses), so we want to return an intersection of these results. To do so, we perform a cartesian product over them (`gen_setX`, defined in Figure 10.8), followed by some technical manipulations found in Figure 10.19 for type coherence. We illustrate this process *via* a semi-formal example, where DNFs and their implementation as sets of sets are interchangeable.

Figure 10.20 shows two DNFs to merge – the atomic elements, which are sequences in the analysis, are abstracted as their actual nature does not impact the process – whereas Figure 10.21 shows the expected result. The cartesian product, as implemented above, will return a set of tuples of conjunctions when applied to a sequence containing the two DNFs. This structure is shown in Figure 10.22.

We then need to get rid of the *tuple layer* to be left with sets. Function `bigcup_tup` shown in Figure 10.19 merges the sets contained in a tuple. Function `bigcup_cart`, found in the same figure, uses it to return a set of sets as shown in Figure 10.23, which happens to be the encoding of the expected result.

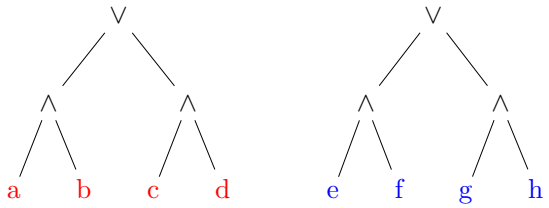


Figure 10.20: Two DNFs to merge

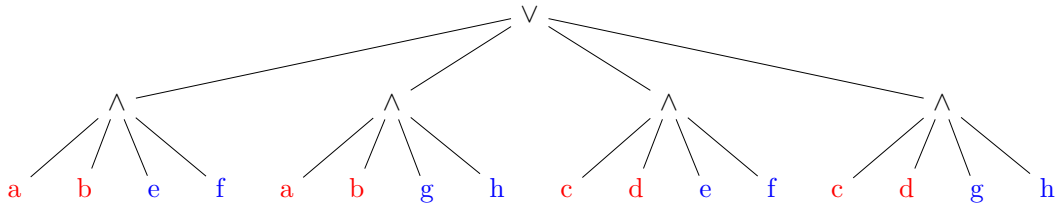


Figure 10.21: Expected results

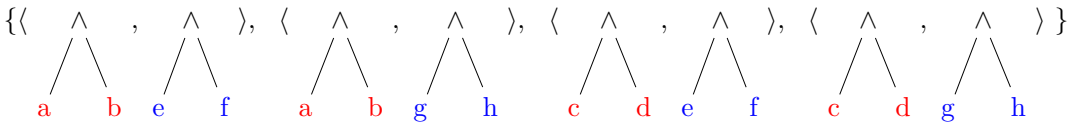


Figure 10.22: Result of the cartesian product

$$\{\{a, b, e, f\},\ \{a, b, g, h\},\ \{c, d, e, f\},\ \{c, d, g, h\}\ \}$$

Figure 10.23: Flattening the cartesian product

Figure 10.24 shows how the analysis is called. Arguments `def` and `dv` are default constants and variables, introduced only for typing purposes, whereas `p` and `v` are the analyzed program and variable. Finally, `#|rul_gr_finType|` is the *fuel* with which the `count` argument is instantiated. The use of this value is discussed in Section 10.4.4.

Now that we have defined an analysis, we need to be able to extract a set of substitutions from it.

```
Definition analysis := analyze_var p (Val def) dv v #|rul_gr_finType|.
```

Figure 10.24: Storing the analysis

10.3.4 From analyses to sets of values

The analysis being formalized as a set of sets of lists (or branches), we define the extraction at all these levels, starting with the lists.

These branches represent, from right to left, paths taken by values from the EDB to the analyzed variable. Before going through the EDB to look for facts, we need to figure out which predicate and argument index to use, i.e. those that correspond to the extensional

predicate at the start of the path. More concretely, we need to be able to extract the predicate and argument index of the last element of the occurrence list, which is done by the functions of Figure 10.25.

```
(* Predicate to which branch [br] leads *)
Definition branch_pred (br : seq (t_occ p)) :=
  match br with
    | [::] => df (* default predicate symbol *)
    | a :: l => match p_at (last a l) with
        | None => df
        | Some f => f end end.

(* Argument index of the last occurrence in the branch *)
Definition branch_t_ind (br : seq (t_occ p)) :=
  match br with
    | [::] => 0
    | a :: l => t_ind (last a l) end.
```

Figure 10.25: Fetching predicate and argument index from analysis branch

Remark 10.15. By construction, the cases than return the assumed default predicate value `df` are never used. Similarly, the `last` function expects a default value to return in case the provided list is empty, but it will not be the case in practice.

Notation 10.16. The functions of Figure 10.25 are defined on sequences of *tocc*s. However, they will be used in practice on the type of branches the analysis returns , i.e. `uniq_seq`s rather than simple lists. As shown in Figure 10.26, we call this type `dbranch`.

Once we have the predicate and argument index of the last element of a branch, we go through the EDB, looking for any fact defined with the predicate, and extract the relevant constant argument, as shown in Figure 10.26. Default constant and predicate values `def` and `df`, introduced above, are once again irrelevant in practice but required.

```
Definition dbranch := (@uniq_seq_finType (t_occ_finType p)).

Definition extract_vals_br (br: dbranch p) : {set syntax.constant} :=
  [set (nth def (arg_gatom f) (branch_t_ind br))
         | f : gatom in i & sym_gatom f == branch_pred df br].
```

Figure 10.26: Extracting values from a branch

Finally, and as previously stated, the *inner sets* are seen as set intersections, whereas the outer set is a union. These are simply translated using the `bigcap` and `bigcup` notations of MathComp. Figure 10.27 shows this step and how, once we have extracted the variables of an extraction, we build a set of *singleton substitutions*, i.e. substitutions that only map the studied variables to the different values.

```
Definition extract_vals_conj (cj : {set dbranch p}) : {set syntax.constant} :=
  \bigcap_(br in cj) extract_vals_br br.

Definition extract_vals_disj (disj : {set {set dbranch p}}) : {set syntax.constant} :=
  \bigcup_(cj in disj) extract_vals_conj cj.

Definition extract_vals_sub : {set sub} :=
  [set (add emptysub v c) | c in extract_vals_disj analysis].
```

Figure 10.27: From branches to substitutions

Although small in size and despite the heavy use of set notations, the definition of the static analysis, and especially the implementation of the analysis itself, might seem surprisingly convoluted to the reader, considering how simple the base idea driving it is. This oddity – which appeared to us early on – motivated us to verify the whole optimization process in Coq. The next section introduces the certification of the static analysis.

10.4 Certification

The certification of the static analysis requires the introduction and formalization of a new tool we call the *no-recursion trace*. This section first explains why it is required and provides an intuition of its nature. We then formalize it, which allows its use in the verification of the static analysis. After discussing that general process, we focus on a specific aspect, namely the certification of the termination of the analysis.

10.4.1 Another angle on traces

As previously stated, the analysis does not consider the same program point twice, allowing a fast and terminating analysis of recursive programs. However, it also makes it harder to relate the produced tree and the actual deduction of facts, i.e. traces. To do so, we introduce an intermediate layer, which we call the *no-recursion trace*. The idea is, given a deduction, to identify a truncated version of the trace that is closer to the analysis while still preserving enough information to be related to the actual trace semantics – a problem already tackled in other verification contexts [Jeannet and Serwe, 2004, Shivers, 1991].

The truncation of the trace is done at the level of branches. More concretely, given a deduction trace and a variable, we look at the branches of the former, which all correspond to a path followed by a value from the EDB to the variable, and transform them into truncated, repetition-free sequences of *tocc*s that approximate the aforementioned path. The no-recursion trace is then presented as a set of `uniq_seq`s, as introduced in Section 7.1.2.

Example 10.17. We reuse the graph connectivity program from Example 2.9 and the trace of Figure 8.1c, which represents a deduction of $path(4, 3)$. Let us compute the no-recursion trace of X in the second rule. X has only one occurrence in the body of the rule, at index 0 of the *path* atom. We then look at the child corresponding to the atom – the left child – of the actual trace, which contains the same clause. The corresponding term, i.e. the one at index 0 of the head of the clause, is again X, which only occurs in the *path* predicate.

Since this *tocc*, $\langle 1, 0, 0 \rangle$, has already been visited, it is not added again to the sequence. In

a sense, we ignore this step and keep exploring the trace, which leads us to the first clause. We add the *tocc* of X in the *edge* predicate, i.e. $\langle 0, 0, 0\rangle$. The next child in the trace is a leaf (the *edge* predicate is extensional), so we stop here and return a set that only contains the sequence $[\langle 1, 0, 0\rangle, \langle 0, 0, 0\rangle]$. This sequence indeed is a truncated, repetition-free version of the *flow* taken by values from the EDB to the variable w.r.t. the actual deduction, or trace, of $path(4, 3)$.

Remark 10.18. In spite of its name, and as seen in Example 10.17, the no-recursion trace can capture executions which do use recursive rules. Moreover, the returned lists have the unicity property, but can contain multiple *tocc*s which refer to the same clause. Example 12.3 will illustrate this point.

10.4.2 Formalization of the no-recursion trace

The Coq definition of the no-recursion trace is shown in Figure 10.28. The `tr` argument is the trace to be truncated, `prev` is the set of previously visited program locations, `v` the variable to focus on, and `count` is used to ensure the termination of the function. Given a strictly positive value of `count`, we take a look at the shape of the trace. If `tr` is a leaf, we return a set only containing an empty list, which will be enriched at the level of nodes.

```
Fixpoint unrec_trace_gen (prev : {set tocc}) (tr : trace) (v : var)
                         (count : nat) : {set (uniq_seq tocc)} :=

  (* Ensuring termination of the funciton *)
  match count with | 0 => set0 | count.+1 =>

  (* If tr is a leaf, the empty list encapsulated in a set is returned *)
  match tr with
    | ABLeaf _ => [set unil]
    | ABNode (RS cl s) descs =>

        (* Unfolding the next call *)
        let unrec_b (o : t_occ p) : {set dbranch} :=
          match (nth_error descs (b_ind o)) with
            | None => set0 (* None case not used in practice *)
            | Some (ABLeaf _) => [set unil]
            | Some (ABNode (RS clb sb) descsb) =>

                (* recursive call, with o added to prev *)
                unrec_trace_gen (o |: prev)
                                (ABNode (RS clb sb) descsb)
                                (get_cl_var dt dv clb (t_ind o))
                                count end
      in

      (* Computing non-visited occurrences of v *)
      let occs := (occsInProgram p v) :\: prev in

      (* adding occ on top of the returned set of sequences *)
      \bigcup_(occ in occs) [set pucons occ l | l in unrec_b occ]
       end end.
```

Figure 10.28: The no-recursion trace in Coq

Indeed, given a node storing a clause `cl` and a substitution `s`, and whose children are stored in the `descs` list, we first compute the occurrences of `v` that have not been visited yet. Then, for each such occurrence `o`, we perform a recursive call on the i^{th} child, where i is the atom index within the body (`b_ind`) of the given occurrence. However, since the function takes the currently variable as an argument, we need to be able, when performing the recursive call, to provide it, and thus to have access to the next clause the trace leads us to.

To circumvent this issue, we partially unfold the recursion in the `unrec_b` function. If the i^{th} child is a leaf, then, in concordance the general definition, we return a singleton containing the empty list. Otherwhise, we get access to the new clause stored in the child node, called `clb`. We then get the variable at the j^{th} position of its head, where j is the argument index of `o`. After that – and adding `o` to `prev`, the recursive call is performed.

When recursion w.r.t. an occurrence returns a set of `uniq_seq`s, we add that occurrence on top of all of them, using the `pucons` function defined in Figure 7.11. Finally, the set union of all such sets is returned.

We can now use this no-recursion trace to bridge the gap between the actual traces and the analyses we produce.

10.4.3 Relating executions and analyses

The first step of the certification is to state, and prove, the completeness of the no-recursion trace. More concretely, for every trace and involved variable, every branch in the no-recursion version of the trace must lead to a fact in the EDB that is relevant to the execution.

```
(* Characterization of the adequacy of a dbranch w.r.t.
   a substitution, a variable and an interpretation *)
Definition br_adequate def (br : dbranch) (s : sub)
                           (v : 'I_n) (i : interp) : bool :=
  [exists c : syntax.constant, (s v == Some c) &&
    [exists ga in i, (sym_gatom ga == branch_pred br)
                  && (nth def (arg_gatom ga) (branch_t_ind br) == c)]].

Theorem no_rec_needed tr v (i : interp) (m : nat)
                      (cl : clause) (s : sub) :

   (* tr is a trace obtained with interpration i *)
   tr \in sem_t p def m i
   (* whose root is cl,s *)
-> ABroot tr = inl (RS cl s)
   (* v is a variable of cl *)
-> v \in tail_vars (body_cl cl)
   (* each dbranch of the no-recursion trace
      is adequate w.r.t. s, v and i *)
-> [forall br in unrec_trace tr v (height tr).+1,
      br_adequate def br s v i].
```

Figure 10.29: Completeness of no-recursion traces

Even more concretely, for every trace tr leading to a deduction combining a clause C with a

substitution σ, and every variable v occurring in C, any branch in the no-recursion version of tr w.r.t. v leads to a *tocc* $\langle x, y, z\rangle$ that corresponds to an extensional predicate p such that the EDB contains a fact f whose z^{th} argument is the $\sigma(v)$ constant. Figure 10.29 shows the Coq definitions and the exact formulation of the completeness.

In other words, the no-recursion trace is used to compute the bounded part of the constraints which the actual semantics enforces, and, in that sense, it is complete. Another interpretation is that, as intuited by the theorem's name, dealing with recursion (see Remark 10.18) is not necessary to get a first approximation of the semantics of a program.

Now that we related the actual traces and their no-recursion counterparts, we need to do the same with the latter and the static analysis. This is where the encoding of the analyses as sets of sets of branches comes in handy, as the core lemma simply states that the no-recursion trace is actually and simply an element of the analysis, meaning that it is captured. This lemma is shown in Figure 10.30.

Combining the results of Figures 10.29 and 10.30 would obviously be the key ingredient to prove that the static analysis captures (and may approximate) the semantics of Datalog. However, doing so would leave a blindspot, which is the termination of the analysis.

```
Lemma no_rec_capt prev tr i m cl s v :
   tr \in sem_t p gat_def def m i
-> ABroot (val tr) = inl (RS cl s)
-> v \in tail_vars (body_cl cl)
-> unrec_trace_gen prev tr v (height tr).+1
 \in analyze_var_prev prev v (height tr).
```

Figure 10.30: No-recursion trace capture

10.4.4 Termination of the analysis

Lemma `no_rec_capt` uses `analyze_var_prev` with its fuel argument, `count`, set to the height of the considered trace. However, the height of traces is theorically not bounded (see for example the program of Figure 10.4a, where one may use the first rule an arbitrary number of times), meaning that this lemma does not provide a satisfiable guarantee that the analysis is usable in practice. We then need to find a bounded value for `count` and show that its use preserves the completeness result.

Our first intuition for its concrete value was the cardinal of the *tocc* type, i.e. the number of arguments positions within the bodies of the given program. Although it probably could have easily been shown to be an adequate value, a different, potentially surprising answers emerges from `no_rec_capt` and our development of finite tree types.

Indeed, as stated in Section 8.3, we implement the traces as `WUtree rul_gr gatom`. Lemma `height_WUtree`, defined in Figure 7.27, then implies that the height of any trace we consider is bounded by `#|rul_gr_finType|`, the cardinal of the `rul_gr` type. Morever, Lemma `trace_sem_completeness` stated in Figure 8.12 shows that restricting ourselves to these bounded traces is sufficient to capture the full semantics of Datalog. Thanks to these results, and after showing the monotonicity of the analysis w.r.t. its `count` argument, we can define and certify an execution of the static analysis with a program-specific bound, as seen in Figure 10.31.

```
Lemma analyze_incr prev v (m1 m2 : nat) :
   m1 <= m2
-> analyze_var_prev prev v m1 \subset analyze_var_prev prev v m2.

Theorem no_rec_capt_nf tr i m cl s v :
   tr \in sem_t p gat_def def m i
-> ABroot (val tr) = inl (RS cl s)
-> v \in tail_vars (body_cl cl)
-> unrec_trace p tr v (height tr).+1
 \in analyze_var v #|rul_gr_finType|.
```

Figure 10.31: No-recursion trace is captured by a bounded analysis

Remark 10.19. The actual value of `#|rul_gr_finType|` is the number of rules of the considered program multiplied by the cardinal of the substitution type, i.e. $(c+1)^v$, where c and v are the number of constants and variables, respectively (the $+1$ takes into account the fact that substitutions are partial functions, and thus may associate no value to some variables). This number is obviously much higher than the number of *tocc*s in most programs[2], but it is actually irrelevant. In practice, the analysis will stop when there is no new *tocc* to explore in each branch. What Theorem `no_rec_capt_nf` shows is that this will happen at some point.

Remark 10.20. Some alternative strategies can be used to show the termination of complex functions without the use of fuel – which happens to *pollute* the extracted code –, such as the Braga method [Larchey-Wendling and Monin, 2018]. However, the goal of our work was simply to validate an optimization scheme before implementing it in a Python-based project (see Chapter 6), rather than extract it in the form of OCaml code. Due to this point, and timing issues, we did not consider these more complex methods.

10.4.5 Value and substitution extraction

We showed in Section 10.3.4 how we implement the extraction of substitutions from a given analysis. The certification of this code is in three steps. First, we provide an alternative definition of the extraction, shown in Figure 10.32. This definition, although *de facto* computable, is rather used as a specification.

```
Definition extract_subs_spec : {set sub} :=
 [set s : sub | (dom s == [set v])
             && [exists ct in analysis,
                   [forall br in pred_of_set ct,
                      @br_adequate p df def br s v i]]].
```

Figure 10.32: Specification of the extraction

The use of `br_adequate` in this definition makes it easier to relate it to the results introduced in Section 10.4.3. More concretely, the second step of the proof is to combine this new

[2] In fact, the unicity of variables across rules makes that inequality hold in general. Without this hypothesis, one can come up with programs where it is reversed

definition with Theorem `no_rec_needed` (cf. Figure 10.29) to show that the analysis can in theory be used to extract a set of substitutions that overapproximates the semantics of the studied program, w.r.t. Definition 9.4, as shown in Figure 10.33.

```
Lemma static_extract_spec :
  [forall cl in p,
    (0 < #|tail_vars (body_cl cl) :&: [set v]|)
==> [forall s, bmatch def (ffp p i def) cl s
==> (sub_filter s [set v] \in extract_subs_spec)]].
```

Figure 10.33: Completeness of the specification of the extraction

The third and final step is to show the equivalence of the two definitions – the specification and the actual implementation – of the extraction. Once we have this result, it can be combined with `static_extract_spec` to show that the program partial instance described in Chapter 9 used with the static analysis indeed preserves the semantic of the program. These results are shown in Figure 10.34.

```
Lemma extract_vals_sub_adequate :
  extract_vals_sub = extract_subs_spec.

Theorem static_extraction_adequacy (m : nat):
  (sem (@tprog p [set v] extract_vals_sub) def m i) = (sem p def m i).
```

Figure 10.34: Adequacy of the partial instance with the static analysis

Remark 10.21. The reader may notice a discrepancy between the analysis and the rewriting, as the former fundamentally provides values for only one variable, and the latter assumes a set of substitutions, i.e. potentially instantiates multiple variables at once. The extraction of values from an analysis does return a set of substitutions (see function `extract_vals_sub` in Figure 10.27), but they map only one variable to an actual value, meaning that this is an *artificial* step to fit into the formalism used by the partial program instance of Chapter 9.

In practice, to instantiate multiple variables in a single rule using the previous definitions, one can either apply the analysis and rewriting multiple times with a substitution on a single variable, or generate substitutions using a cross product of the different value sets. These two methods produce the same programs, as well as the same (still) inefficient results. Section 12.2 adresses this performance issue, by proposing and discussing an improved version of the static analysis from which one can actually extract actual, multi-variable substitutions.

Chapter 11

Predicate specialization

C'est en 1776 que survint la dernière
métamorphose de Joseph Curwen

H.P. Lovercraft, *L'affaire Charles Dexter Ward*,
traduit de l'anglais par Arnaud Demaegd

The second program transformation we introduce aims at the reduction of the sizes of the tables used in a Datalog engine. To do so, it introduces new predicates to partition existing relations into smaller ones. Section 11.1 first presents the general idea of the transformation, then Section 11.2 formalizes and justifies it. Finally, Section 11.3 presents the implementation of the rewriting in Coq.

11.1 Intuition

The transformation assumes an intensional predicate p such that one of its arguments is always a constant in the rules defining it. This is a very simple and syntactic criteria to determine a subset – technically an overapproximation – of the constants that can be found during the execution of the program in deduced facts about p at the corresponding predicate and index.

Example 11.1. Figure 11.1 shows a program fragment. The first two rules define a predicate p of arity 3, and the third and fourth rules use it. Assuming the full program contains no other rule defining p, and considering the separation between extensional and intensional predicates (see Section 2.1.2), we can be sure that the first argument of any deduced p fact will be 1 or 2.

$$
\begin{array}{ll}
p(1, Y, Z) & \leftarrow q(Y, Z)\,. \\
p(2, Y, Z) & \leftarrow r(Z, Z, Y)\,. \\
t(X) & \leftarrow p(1, X, X)\,. \\
u(X) & \leftarrow p(X, X, X)\,.
\end{array}
$$

Figure 11.1: Defining and using p

Assuming such a setting, the predicate with a clearly bounded argument can be replaced by a set of specialized versions, one for each identified relevant value. We introduce two new predicates, called p_1 and p_2, of arity 2. These predicates are meant to be specialized version of p, where the first argument does not explicitely appear but is implicitely considered to be 1 or 2. The rules of Figure 11.1 can then be replaced by those of Figure 11.2.

$$
\begin{array}{ll}
p_1(Y, Z) & \leftarrow q(Y, Z)\,. \\
p_2(Y, Z) & \leftarrow r(Z, Z, Y)\,. \\
t(X) & \leftarrow p_1(X, X)\,. \\
y(X) & \leftarrow p(X, X, X)
\end{array}
$$

Figure 11.2: Specialized program

This requires another modification to work, as we no longer deduce facts about p, and yet still have an occurrence of p in the body of the fourth rule. To allow the use of rules that contain non-specialized p atoms in their body, we need to add the rules of Figure 11.3 to the new program. The reverse rules (normal to specialized version) are not required, as all the relevant specialized versions of the predicate are already defined by the transformed rules.

$$
\begin{array}{l}
p(1, Y, Z) \leftarrow p_1(Y, Z)\,. \\
p(2, Y, Z) \leftarrow p_2(Y, Z)\,.
\end{array}
$$

Figure 11.3: Relating normal and specialized definitions

Remark 11.2. As previously stated, looking for predicates with arguments that are *statically defined* is a very simple, if not simplistic, way to determine a set of values that overapproximates the behavior of a predicate's argument. It might even feel suspicious to rely on this shallow method when the previous chapter introduced a static analysis that computes a similar information in a less trivial way.

As discussed in Section 12.1, Octant uses the predicate specialization on top of the partial program instance of Chapter 9, which relies on the static analysis of Chapter 10 to produce predicates with arguments that are *statically defined*. In other words, in practice, the predicate specialization leverages the static analysis *via* the partial instance.

11.2 Formalization and justification

The building block of this program transformation is the following definition.

Definition 11.3. **(*spec* – atom specialization)** Given a predicate symbol p, an index i, and an atom a, the *spec* atom specialization function is defined as

$$
spec(a, p, i) = \begin{cases} p_c(t_1, ..., t_{i-1}, t_{i+1}, ..., t_{arity(p)}) \\ \qquad \text{if } a \text{ is of the form } p(t_1, ..., t_{i-1}, c, t_{i+1}, ..., t_{arity(p)}) \text{ with } c \text{ a constant} \\ a \\ \qquad \text{if } a \text{ is of the form } q(t_1, ..., t_{i-1}, t_i, t_{i+1}, ..., t_{arity(q)}) \text{ with } t_i \text{ a variable or } q \neq p \end{cases}
$$

Notation 11.4. In the rest of this section, we assume a Datalog program P where the i^{th} argument of an intensional predicate p of arity m is always defined and no predicate of the form p_c appears. The *spec* function is lifted to clauses (heads and bodies) and programs, and the transformed program will be denoted as $spec(P, p, i)$. Although it is not formally implied by the definition of *spec*, for clarity purposes (i.e. reduce notations), we will consider that the rules such as those of Figure 11.3 are packaged in $spec(P, p, i)$.

To certify the transformation, we first need to address the formulation of the targeted adequacy results. Theorem 11.5 shows the targeted completeness result. The number of steps used in the transformed program is doubled, to allow the use of rules such as those shown in Figure 11.3.

Theorem 11.5. (Predicate specialization completeness)
For any number of steps k, $(T_P \uparrow k)(I) \subseteq (T_{spec(P,p,i)} \uparrow 2k)(I)$.

The transformed program produces specialized facts that did not appear in the original program, which is not a concern for the completeness theorem, as the new facts are on the right side (both literally and figuratively). In comparison, the soundness must be formulated *modulo* those new facts, as in Theorem 11.6.

Theorem 11.6. (Predicate specialization soundness)
For any number of steps k, $\{x \in (T_{spec(P,p,i)} \uparrow k)(I) \mid x \text{ is not specialized}\} \subseteq (T_P \uparrow k)(I)$.

Remark 11.7. One might expect that, in Theorem 11.6, the number of steps used in the transformed program might again be doubled compared to the original one, as in Theorem 11.5, but that would not account for the normal, not specialized part of the program.

The strategy is of course to prove these results by induction on the number of steps k. However, given the formulations above, the induction hypotheses would not be strong enough. For example, the completeness theorem does not say anything about the new, specialized facts, which may be required in the execution of the new program. We then first prove the two following alternative lemmas.

Lemma 11.8. (Strong predicate specialization completeness)
For any number of steps k,
$(T_P \uparrow k)(I) \cup \{spec(a, p, i) \mid a \in (T_P \uparrow k)(I)\} \subseteq (T_{spec(P,p,i)} \uparrow 2k)(I)$.

Remark 11.9. When applied to a ground atom whose predicate symbol is not p, $spec(a, p, i)$ simply returns a. It is then safe to apply the *spec* function to the entirity of $(T_P \uparrow k)(I)$ rather than specifically the relevant facts.

Proof. We proceed by induction on the number of steps k. In the base case, the definitions imply that $(T_P \uparrow 0)(I) = (T_{spec(P,p,i)} \uparrow 0)(I) = I$. The goal then becomes

$$I \cup \{spec(a, p, i) \mid a \in I\} \subseteq I$$

Since p is assumed to be an intensional predicate, there is no fact about it in the EDB I. We then have $\{spec(a, p, i) \mid a \in I\} = I$, which makes the goal for the base case trivial.

In the recursive case, let f be a fact in $(T_P \uparrow k+1)(I) \cup \{spec(a, p, i) \mid a \in (T_P \uparrow k+1)(I)\}$.

We split the proof in three cases, depending on the predicate symbol of f. This disjunction determines which side of the union f belongs to.

f is a p fact

The *spec* function transforms any ground atom about p into a p_c fact, meaning that $\{spec(a, p, i) \mid a \in (T_P \uparrow k+1)(I)\}$ does not contain such an atom. This in turn implies that f, in the current hypothesis, has to be in $(T_P \uparrow k+1)(I)$. In that setting, f was either already in the previous iteration $(T_P \uparrow k)(I)$, or it has just been deduced.

f previously deduced

In this scenario, f is also in $(T_{spec(P,p,i)} \uparrow 2k)(I)$ (induction hypothesis), and thus in $(T_{spec(P,p,i)} \uparrow 2(k+1))(I)$ by monotonicity of the T_P operator.

f just deduced

We can extract a clause C from P and a substitution ν such that ν matches C w.r.t. $(T_P \uparrow k)(I)$.

Let us write $f = p(c_1, ..., c_m)$, where every c_k is a constant (f is a ground atom). Since f is a fact about p, we know that C is headed the same predicate symbol. Then our hypothesis on p implies that its head is of the form $p(t_1, ..., t_{i-1}, c_i, t_{i+1}, ..., t_m)$. In that setting, f is deduced at step $2(k+1)$ of the transformed program using the added rule

$$p(X_1, ..., X_{i-1}, c_i, X_{i+1}, X_m) \leftarrow p_{c_i}(X_1, ..., X_{i-1}, X_{i+1}, ..., X_m).$$

with a substitution that maps X_k to c_k for every k. We then need to show that $p_{c_i}(c_1, ..., c_{i-1}, c_{i+1}, ..., c_m$ is deduced at step $2k+1$.

This deduction is performed using rule $spec(C, p, i)$ with substitution ν. The recursion hypothesis ensures that every ground atom in the body of $\nu(spec(C, p, i))$, even those of the form $p_{c'}(...)$[1], is in $(T_{spec(P,p,i)} \uparrow 2k)(I)$.

f is a p_c fact

Predicates of the form p_c are assumed not to appear in the original program. In this configuration, ground atom f can not be in $(T_P \uparrow k+1)(I)$, meaning that it is in $\{spec(a, p, i) \mid a \in (T_P \uparrow k+1)(I)\}$. Then we can extract a fact g from $(T_P \uparrow k+1)(I)$ such that $f = spec(g, p, i)$, which implies that g is of the form $p(c_1, ..., c_{i-1}, c, c_{i+1}, ..., c_m)$.

This new fact g was either just deduced or already present. In the second scenario, the recursion hypothesis and the monotony of the fixpoint operator show our goal. Otherwise, it was deduced in the original program P using a clause $C \in P$ with a substitution ν.

Fact f is then deduced in the transformed program at step $2k+1$ using the clause $spec(C, p, i)$ with the same substitution. The recursion hypothesis ensures that every ground atom in the body of this clause is in $(T_{spec(P,p,i)} \uparrow 2k)(I)$. The fact is then preserved from $(T_{spec(P,p,i)} \uparrow 2k+1)(I)$ to $(T_{spec(P,p,i)} \uparrow 2(k+1))(I)$.

[1] This is where the induction hypothesis of the lemma's straightforward but weak formulation falls short.

f is a fact about another predicate symbol

The predicate symbol of atom f is different from p, meaning that $spec(f, p, i) = f$. Then, our working hypothesis $f \in (T_P \uparrow k{+}1)(I) \cup \{spec(a, p, i) \mid a \in (T_P \uparrow k{+}1)(I)\}$ is equivalent to simply $f \in (T_P \uparrow k + 1)(I)$.

The rest of the proof is familiar: if f was already in $(T_P \uparrow k)(I)$, we obtain our goal with the induction hypothesis and monotonicity of the fixpoint operator. Otherwise, we extract a clause C and a substitution ν, and use $spec(C, p, i)$ with ν to deduce f.

□

Theorem 11.5 is then a corollary of Lemma 11.8.

As for the soundness, the formulation of Theorem 11.6 considers the specialized facts in a negative way, in the sense that it removes them from the semantics of the transformed program, whereas we need to reason about their presence. We then introduce the following soundness result.

Lemma 11.10. (Strong pred. specialization soundness) For any number of steps k, $(T_{spec(P,p,i)} \uparrow k)(I) \subseteq (T_P \uparrow k)(I) \cup \{spec(a, p, i) \mid a \in (T_P \uparrow k)(I)\}$.

Proof. We proceed by induction on the number of steps k. In the base case, the definitions imply that $(T_P \uparrow 0)(I) = (T_{spec(P,p,i)} \uparrow 0)(I) = I$. The goal then becomes

$$I \subseteq I \cup \{spec(a, p, i) \mid a \in I\}$$

which is automatically true.

In the recursive case, let f be a fact in $(T_{spec(P,p,i)} \uparrow k + 1)(I)$. We split the proof in three cases, depending on the predicate symbol of f.

f is a p fact

Due to the hypothesis on the rules defining p in the P program, every corresponding rule in $inst(P, p, i)$ is headed by a predicate of the form p_c. The only rules of the transformed program that can deduce facts about p are the "specialized to general" rules, such as those of Figure 11.3. This means that f has been deduced *via* a rule of the form

$$p(X_1, ..., X_{i-1}, c_i, X_{i+1}, ..., X_m) \leftarrow p_{c_i}(X_1, ..., X_{i-1}, X_{i+1}, ..., X_m)$$

Writing f as $p(c_1, ..., c_m)$, this means that $p_{c_i}(c_1, ...c_{i-1}, c_{i+1}, ..., c_m) \in (T_{spec(P,p,i)} \uparrow k)(I)$. Combined with the induction hypothesis, we obtain that

$$p_{c_i}(c_1, ...c_{i-1}, c_{i+1}, ..., c_m) \in (T_P \uparrow k)(I) \cup \{spec(a, p, i) \mid a \in (T_P \uparrow k)(I)\}$$

The original program P does not deduce facts about predicates of the form p_c, meaning that

$$p_{c_i}(c_1, ...c_{i-1}, c_{i+1}, ..., c_m) \in \{spec(a, p, i) \mid a \in (T_P \uparrow k)(I)\}$$

We can extract a fact $g \in (T_P \uparrow k)(I)$, st $p_{c_i}(c_1, ...c_{i-1}, c_{i+1}, ..., c_m) = spec(g, p, i)$, which means that $g = p(c_1, ...c_{i-1}, c_i, c_{i+1}, ..., c_m) = f \in (T_P \uparrow k)(I)$. By monotonicity of T_P, f is preserved in $(T_P \uparrow k + 1)(I)$.

f is a p_c fact

Any deduction in the transformed program of a fact whose predicate is of the form p_c is done *via* a rule of the form $spec(C, p, i)$, with C a rule of the original program P, a substitution ν, and every ground atom in the body of $\nu(spec(C, p, i))$ is in the interpretation $(T_{spec(P,p,i)} \uparrow k)(I)$.

Consider C and ν in the original program. We want to show that every ground atom in the body of $\nu(C)$ is in $(T_P \uparrow k)(I)$. Since we are considering a clause of the original program, none of these atoms carries a predicate symbol of the form p_c.

Given such an atom $a \in body(C)$, if it is defined using a predicate symbol that is not p, or its i^{th} argument is a variable rather than a constant, then $spec(a, p, i) = a$. Since $body(\nu(spec(C, p, i)) \in (T_{spec(P,p,i)} \uparrow k)(I)$, so is $\nu(a)$. Using the induction hypothesis, we then obtain that $\nu(a) \in (T_P \uparrow k)(I)$.

On the other hand, if a is of the form $p(t_1, ..., t_{i-1}, c, t_{i+1}, ..., t_m)$ and $\nu(t_j) = c_j$ for every j between 1 and m, then

$$spec(\nu(a)) = p_{c_i}(c_1, ..., c_{i-1}, c_{i+1}, ...c_m) \in (T_{spec(P,p,i)} \uparrow k)(I)$$

and (recursion hypothesis)

$$(T_{spec(P,p,i)} \uparrow k)(I) \subseteq (T_P \uparrow k)(I) \cup \{spec(a, p, i) \mid a \in (T_P \uparrow k)(I)\}$$

Since $(T_P \uparrow k)(I)$ does not contain ground atoms about predicates of the form p_c, we know that $spec(\nu(a)) \in \{spec(a, p, i) \mid a \in (T_P \uparrow k)(I)\}$, which in turns shows that $\nu(a) \in (T_P \uparrow k)(I)$.

Every ground atom in the body of $\nu(C)$ is in $(T_P \uparrow k)(I)$, meaning that we can use C and ν in the original program to deduce the *non-specialized version* of f at step $k + 1$. Then, $f \in \{spec(a, p, i) \mid a \in (T_P \uparrow k + 1)(I)\}$, which implies our goal.

f is a fact about another predicate symbol

In this case, f is deduced in the transformed program using a rule $spec(C, p, i)$, whose head did not change after the application of the *spec* function, and substitution ν. We use in the original program the C rule and the same substitution. The reasoning regarding the availability of the body of $\nu(C)$ is the same as in the previous case, but the deduced fact is directly f and in $(T_P \uparrow k + 1)(I)$.

□

Theorem 11.6 is easily proved using the previous lemma. We can now move on to the formal definition and proof of these results.

11.3 Coq implementation

The idea behind this optimization is rather simple, and although the adequacy proofs require the consideration of many technical cases, they do not contain any fundamental difficulty. In

comparison, the Coq implementation is not as straightforward. In particular, the introduction of new, specialized predicates and the use of the *relating rules*, such as seen in Figure 11.3, require some hypotheses and machinery, as presented in this section.

11.3.1 Hypotheses on the program and specialization

Let us assume a safe (see Figure 3.10) Datalog program P, which contains an intensional predicate f whose ind^{th} argument of a predicate is always a constant. Figure 11.4 shows the Coq implementation of these hypotheses.

```
Variable p : program.
Hypothesis psafe : prog_safe p.

(* [i] is an initial interpretation with only extensional predicates *)
Variable i : interp.
Hypothesis isafe : safe_edb i.

(* default constant, required throughout the formalization *)
Variable def : syntax.constant.

(* [f] is an intensional predicate whose [ind]th index is always a constant *)
Variable f : symtype.
Hypothesis ftype : predtype f = Idb.
Variable ind : 'I_(arity f).

Hypothesis always_cons :
  [forall cl in p, ((hsym_cl cl) == f) ==>
    [exists c:syntax.constant,
      nth_error (arg_atom (head_cl cl)) ind == (Some (Val c))]].
```

Figure 11.4: Hypotheses on the transformed program

We need a mechanism to build the new, specialized predicates. To do so, we assume a function, named `pclone`, that associates a predicate to any constant. Figure 11.5 shows its type and a characterization of the newly introduced predicates.

```
Variable pclone : syntax.constant -> symtype.

Definition is_clone_pred (g : symtype) : bool :=
 [exists c, g == pclone c].

Definition is_clone_ga (ga : gatom) : bool :=
  is_clone_pred (sym_gatom ga).
```

Figure 11.5: Building and identifying the new predicates

We also need some assumptions on the cloning function. The introduced predicates must be new, in the sense that they do not already appear in the rules or EDB. It must also be

different from f[2], and the cloned predicates must all be different. Finally, we need to relate their arity with that of f. Figure 11.6 shows the Coq definitions of these hypotheses.

```
(* Cloned predicates are entirely fresh *)
Hypothesis pfresh : [forall c, pclone c \notin sym_prog p].
Hypothesis ifresh : ~~ [exists x in i, is_clone_ga x].

(* Cloned predicates are different from [f] and one another *)
Hypothesis pnotf : [forall c, pclone c != f].
Hypothesis pinj  : injective pclone.

(* The arity of cloned predicates is [f]'s minus one *)
Hypothesis parity : [forall c, arity f == (arity (pclone c)).+1].
```

Figure 11.6: Hypotheses on the specialized predicates

Remark 11.11. These hypotheses only concern our Coq formalization, and their enforcement by an implementation of this transformation still has to be checked.

We can now use the predicate specialization function to rewrite the rules of the program.

11.3.2 Rewriting the rules

We proceed incrementally, starting with atoms. Figure 11.7 shows the core function. It takes as arguments an atom a, as well as a (bounded) index j. This index generalizes `ind`, to allow inductions on atoms, or rather the list of arguments of an atom.

```
Definition raw_atom_clone (j : 'I_(arity f)) (a : raw_atom) : raw_atom :=
  match a with
  | RawAtom pr args =>
      if (pr == f) then
        match nth_error args j with
          | Some (Val c) => RawAtom (pclone c) (sremove args j)
          | _ => RawAtom pr args end
      else RawAtom pr args end.

(* try to remove the [i]th element of [s] *)
Fixpoint sremove {A : Type} (s : seq A)  (i : nat) : seq A :=
  match i with
    | 0 => behead s
    | i.+1 => match s with
      | [::] => [::]
      | a::l => a::sremove l i end end.
```

Figure 11.7: Specializing a raw atom

[2]Theoretically, the hypothesis that no cloned predicate appears in the rules implies that they are all different from f. However, Due to Coq's thoroughness, we need to consider the case where f does not appear in the studied program, in which case we lose the implication. We deemed it simpler and more elegant to add the hypothesis of inequality between the cloned predicates and f.

If the atom is an occurrence of f and has a constant c as its $\mathtt{j}^{th}$ argument, we substitute f by f_c and remove the incriminated argument.

As explained in Section 3.2.1, DatalogCert defines raw and ground atom types, and builds the actual atom and ground atom types on top of them, by adding the well-foundeness condition, i.e. ensuring that the number of arguments in the atom matches the arity of the involved predicate. Figure 11.8 shows the additional steps required to lift the specialization from raw atoms to actual atoms and clauses.

```
Lemma wf_clone (j : 'I_(arity f)) (a : atom) :
  wf_atom (raw_atom_clone j a).

Definition atom_clone (j : 'I_(arity f)) (a : atom) : atom :=
  Atom (wf_clone j a).

(* wmap is map for Wlist *)
Definition tail_clone (j : 'I_(arity f)) (tl : tail) : tail :=
  wmap (atom_clone j) tl.

Definition cl_clone (j : 'I_(arity f)) (cl : clause) : clause :=
  match cl with Clause h tl
    => Clause (atom_clone j h) (tail_clone j tl) end.
```

Figure 11.8: Specializing atoms and clauses

The program is not the only place where atoms have to be specialized, as we will need to reason on the semantics of the transformed program, and thus specialize ground atoms. For the sake of exhaustivity, Figure 11.9 shows the simple adaptation of the previously seen functions to this type.

```
Definition raw_gatom_clone (j : 'I_(arity f)) (a : raw_gatom) : raw_gatom :=
  match a with
  | RawGAtom pr args =>
      if (pr == f) then
        match nth_error args j with
          | Some c => RawGAtom (pclone c) (sremove args j)
          | None => RawGAtom pr args end
      else RawGAtom pr args end.

Lemma wf_gclone (j : 'I_(arity f)) (ga : gatom) :
  wf_gatom (raw_gatom_clone j ga).

Definition gatom_clone (j : 'I_(arity f)) (ga : gatom) : gatom :=
  GAtom (wf_gclone j ga).
```

Figure 11.9: Specializing a ground atom

As explained in Example 11.1, on top of the specialization of relevant occurrences of f, we need to add rules that relate them to the original definition. As shown by the next few pages, formalizing and using these rules turned out to be more of a challenge than one might have

expected.

11.3.3 Writing new generic rules

We add one $f \leftarrow f_c$ rule for each constant c used – at the $\texttt{ind}^{th}$ index – in the definition of f. We then first need to collect the values used to define the $\texttt{ind}^{th}$ term of f. This is done using the functions shown in Figure 11.10.

```
Definition ind_terms :=
  pmap (fun cl => nth_error (arg_atom (head_cl cl)) ind)
       [seq cl <- p | hsym_cl cl == f].

Definition ind_vals :=
  pmap (fun t => if t is Val c then Some c else None) ind_terms.
```

Figure 11.10: Computing the approximation of f

Function `ind_terms` filters the clauses that are headed by f (`hsym_cl` is defined in Figure 7.30) and fetches the $\texttt{ind}^{th}$ arguments of the head within an option type using `nth_error`. Since `ind` is defined as being at most the arity of f, this will only return `Some` elements in practice, meaning that we can safely extract the terms using `pmap`. Then, `ind_vals` performs another fictional job for typing purposes, i.e. filtering out the variables which do not happen in practice thanks to the hypothesis on the definition of f.

We now have to manually add the rules to the produced program. As explained in Section 3.2, the atoms carry a proof that their number of arguments is the arity of the associated predicate, and the variables are encoded as ordinals, meaning that we need to deal with a lot of dependent types. This is illustrated by Figure 11.11, which shows how we define the sequence of variables that will serve as a building block of the added rules.

Once again, we generalize `ind` to allow usable inductions in the proofs. Also note that we must ensure that `n` is at least the arity of f, i.e. that there are enough available variables (see Section 3.2.1) to build the rules.

```
Definition dep_iota (m k : nat) : seq ('I_(m+k)) :=
  pmap insub (iota m k).

(** [X_1, X_2, ..., X_j] *)
Definition gen_vars_j (j : 'I_n.+1): seq term :=
 map (fun x => Var x)
     (map (fun x : 'I_j => widen_ord (ltn_ord j) x)
          (dep_iota 0 j)).

Hypothesis arity_vars : arity f < n.+1.

(** [X_1, X_2, ..., X_(arity f)] *)
Definition gen_vars : seq term :=
  gen_vars_j (Ordinal arity_vars).
```

Figure 11.11: Manually defining a sequence of variables

We need to either take away (for the instances of the specialized predicates) or replace by a constant (for the left side of the added rules) a variable in these sequences. Figure 11.12 shows how it is done.

```
(* [X_1, X_2, ..., X_(k-1), X_(k+1), ..., X_(j)]  *)
Definition gen_vars_rem_j (j : 'I_n.+1) (k : 'I_n) : seq term :=
  rem (Var k) (gen_vars_j j).

(* [X_1, X_2, ..., X_(j-1), X_(j+1), ..., X_(arity f)]  *)
Definition gen_vars_rem (j : 'I_(arity f))  : seq term :=
  gen_vars_rem_j (Ordinal arity_vars) (@widen_ord (arity f) n arity_vars j).

(* [X_1, X_2,... X_(ind-1), c, X_(ind+1), ..., X_(arity f)] *)
Definition gen_vars_c_f (j : 'I_(arity f))  (c : syntax.constant) :=
  set_nth (Val c) gen_vars j (Val c).
```

Figure 11.12: Modifying sequences of variables

We can now build the raw and full atoms, showing that the well-foundedness is preserved, as seen in Figures 11.13 and 11.14.

```
(* raw f(X_1, X_2,... X_(j-1), c, X_(j+1), ..., X_(arity f)) *)
Definition raw_gen_c_f (j : 'I_(arity f))  (c : syntax.constant) : raw_atom :=
  RawAtom f (gen_vars_c_f j c).

Lemma raw_gen_f_c_wf (j : 'I_(arity f)) (c : syntax.constant) :
  wf_atom (raw_gen_c_f j c).

(* f(X_1, X_2,... X_(j-1), c, X_(j+1), ..., X_(arity f)) *)
Definition gen_f_c (j : 'I_(arity f))  (c : syntax.constant) : atom :=
  Atom (raw_gen_f_c_wf j c).
```

Figure 11.13: Building generic atoms with a constant argument

```
(* raw f_c(X_1, X_2,... X_(j-1), X_(j+1), ..., X_(arity f)) *)
Definition raw_gen_f_c (j : 'I_(arity f))  (c : syntax.constant) : raw_atom :=
  RawAtom (pclone c) (gen_vars_rem j).

Lemma raw_gen_c_f_c_wf (j : 'I_(arity f))  (c : syntax.constant) :
  wf_atom (raw_gen_f_c j c).

(* f_c(X_1, X_2,... X_(j-1), X_(j+1), ..., X_(arity f)) *)
Definition gen_c_f_c (j : 'I_(arity f))  (c : syntax.constant) : atom :=
  Atom (raw_gen_c_f_c_wf j c).
```

Figure 11.14: Building generic atoms with a constant in the predicate symbol

Moving on to the rules, we need to explicit that `bn`, the maximal length of the bodies of rules,

is strictly positive. As shown in Figure 11.15, this finally allows the definition of our custom clauses, and ultimately, the full tranformation of the program.

```
Hypothesis bn_not_zero : 0 < bn.

Lemma gen_c_f_c_size (j : 'I_(arity f))  (c : syntax.constant) :
 size [:: gen_c_f_c j c] <= bn.

Definition c_to_gen (j : 'I_(arity f)) (c : syntax.constant) :=
  Clause (gen_f_c ind c) (seq_to_wlist_uncut (gen_c_f_c_size j c)).

(* Potentially transformed clauses ++ new specialized to generic rules *)
Definition proj_prog :=
  (map (cl_clone ind) p) ++ [seq c_to_gen ind c | c in ind_vals].
```

Figure 11.15: Building generic rules

Function `seq_to_wlist_uncut` expects a proof that a sequence `l` has a length lower than `k`, and returns `l` seen as a `Wlist k`. It is then used in `c_to_gen` to fit the generic rules into the framework of clauses in DatalogCert.

We can now discuss the certification of this transformation. The biggest challenge lied in the use of the generic rules we just defined, so let us first focus on this point.

11.3.4 Using the generic rules

The point of DatalogCert is the verification of a Datalog engine, i.e. ensuring that iterating an implementation of the T_P operator on any given program will compute the expected semantic. In that sense, the matching mechanism is *verified*, not *used* (see Section 3.3.2.1)). However, to show that the predicate specialization preserves the semantics, we need to manually *trigger* the $f \leftarrow f_c$ rules defined in Figure 11.15, i.e. explicitely provide the substitutions they are instantiated with.

```
(* Enriches [s] to map the variables of [args] to the values of [gargs] *)
Fixpoint extract_sub_seq_c (args : seq term) (gargs : seq syntax.constant)
                           (s : sub) : sub :=
  match gargs with
  | [::] => s
  | x::l =>
      match args with
      | [::] => s
      | Val x'::l' => (extract_sub_seq_c l' l s)
      | Var x'::l' => add (extract_sub_seq_c l' l s) x' x end end.

(* [extract_sub_seq_c] in practice *)
Definition extract_sub_ga (a : atom) (ga : gatom) :=
  extract_sub_seq_c (arg_atom a) (arg_gatom ga) emptysub.
```

Figure 11.16: Computing substitutions for the generic rules

These substitutions are computed by a function, shown in Figure 11.16, that takes a list of variables and a list of terms, and creates a substitution that maps each variable to the value at the corresponding index. In practice, this function is used with deduced specialized facts.

The main difficulty in the certification of this method is that a shift in the list of variables fully changes the extracted substitution, meaning that lemmas on matching using this function could not be proved using straightforward inductions. The multiple technical lemmas then had to be proved using an abstraction of the list of variables.

Figure 11.17 shows some of the main lemmas, and the hypotheses encoding the $[X_1, ..., X_k]$ list of variables. These lemmas all make use the unicity property, which ensures that `extract_sub_seq_c` does not overwrite itself. They also enforce that the elements of the provided list list of terms are all variables, to capture the full provided ground atom. Finally, Lemma `extract_sub_seq_rem_map` uses `find` to relate a variable and constant.

```
(* The extraction of [lc] via [lt] applied to [lt] returns [lc] *)
Lemma extract_sub_seq_map (lt : seq term) (lc : seq syntax.constant) :
   uniq lt
-> size lt = size lc
-> [forall t in lt, exists v, t == Var v]
-> lc
 = [seq gr_term_def def (extract_sub_seq_c lt lc emptysub) i0 | i0 <- lt].


(* Extraction s(a2) via a1 applied to a1 returns s(a2) *)
Lemma extract_gr_v (s : sub) (a1 a2 : atom) :
   sym_atom a1 = sym_atom a2
-> [forall t in (arg_atom a1), exists v:'I_n, t == Var v]
-> uniq (arg_atom a1)
-> gr_atom_def def s a2 =
   gr_atom_def def (extract_sub_ga a1 (gr_atom_def def s a2)) a1.


(* Similar to the previous lemmas, but handles the
   removal/addition of constant c *)
Lemma extract_sub_seq_rem_map (lt : seq term) (v : 'I_n) (lc : seq syntax.constant)
                              (j : 'I_n) (c : syntax.constant) :
   j < size lt
-> size lt = size lc
-> uniq lt
-> [forall t in lt, exists vb, t == Var vb]
-> find (fun y => y == Var v) lt = j
-> nth_error lc j = Some c
-> lc = [seq gr_term_def def
             (extract_sub_seq_c (rem (Var v) lt) (sremove lc j) emptysub) i
         | i <- set_nth (Val c) lt j (Val c)].
```

Figure 11.17: Certification of the use of the generic rules

Due to the heavy use of dependent types, the actual proofs of these lemmas are rather technical, and their statements remain quite circumvoluted. Defining and certifying this

part of the rewriting felt like working *against* DatalogCert rather than with it. It is unclear whether our method – both the computation of substitutions or the abstractions used in its certification – can be made simpler. However, in its current state, these functions and results can be used if other program transformations that add rules are to be introduced and verified in the future.

Now that we have the definitions and technical lemmas for all the components of the transformation, we can conclude the adequacy proof.

11.3.5 Combining the pieces

As explained in Section 11.2, the formulation of Theorems 11.5 and 11.6 does not allow a powerful enough induction principle, which led us to introduce Lemmas 11.8 and 11.10. Figures 11.18 introduces the Coq formalization of these intermediate lemmas.

```
Lemma proj_completeness_u (m : nat) :
  sem p def m i :|: [set gatom_clone ind ga | ga in sem p def m i]
    \subset sem proj_prog def m.*2 i.

Lemma proj_soundness_u (m : nat) :
  sem proj_prog def m i \subset
  sem p def m i :|: [set gatom_clone ind ga
                       | ga in sem p def m i & sym_gatom ga == f].
```

Figure 11.18: Intermediate completeness and soundness results

Remark 11.12. The soundness one introduces a `& sym_gatom ga == f` filter that was not present in the paper version above. This condition is theoretically useless, as ground atoms not headed by f are not transformed by `gatom_clone` and already appear on the left side of the set union `:|:`. However, it makes easier the use of one of our technical lemmas, hence its addition.

With these new inductions and the previously presented results, we prove the results of Figure 11.19.

```
Theorem proj_completeness (m : nat) :
  sem p def m i \subset sem proj_prog def m.*2 i.

Theorem proj_soundness (m : nat) :
  [set x in sem proj_prog def m i | ~~ is_clone_ga x ] \subset sem p def m i.
```

Figure 11.19: Final completeness and soundness results

Chapter 12

Discussion and related works

> J'ai souvent l'impression que c'est là le principal travail du détective : effacer les faux départs et toujours recommencer
>
> Agatha Christie, *Mort sur le Nil*,
> traduit de l'anglais par Elise Champon et Robert Nobret

> Il lui fallait en tout cas agir comme un enquêteur, c'est-à-dire récapituler, déduire, induire, déceler des logiques souterraines, laisser les lignes de force émerger. S'il y en avait.
>
> Christian Garcin, *Le bon, la Brute et le Renard*

The contributions introduced in this thesis are the results of three years that have been filled with surprises, errors and doubts – and, every now and then, a suitable idea. This chapter dwelves a bit into this process, as it tries to convey some questions raised by the work on this thesis and explain the answers we provide.

Octant, the network verification programs it contains and the limitations of the underlying Datalog engine are the starting point of our work, as well as our *reference point*. It is then fitting that we open this chapter with Section 12.1, which outlines the effects of our optimizations in the context of Octant. Then, Section 12.2 introduces an alternative definition of the extraction process within our static analysis, which leads to more precise but unsafe results for the analysis of multiple variables in a single rule. Then, Section 12.3 discusses various modelization choices, both at the paper and Coq levels. Finally, Section 12.4 tries to quantify the general proof effort, and Section 12.5 discusses related works, focusing on static analyses of logic programs, dealing with recursion in the analysis of programs in general, and the certification of static analyses.

12.1 Effects of the rewritings in the context of Octant

As explained in Sections 5.3 and 6.2, the Network Optimized Datalog engine uses a representation called *Differences of Cubes*, which does not fare well with some primitive predicates. We did not come up with a formal characterization of this class of primitives, but Section

6.2.1 should give the intuition that the Differences of Cubes representation does not handle well dependencies across the encodings of multiple variables. Focusing on Example 6.6, one might want to be able to write

$$\underbrace{\star\star\star\star}_{v_1}\ \underbrace{\star\star\star\star}_{v_2} \setminus \{\underbrace{0\star\star\star}_{v_1}\underbrace{1\star\star\star}_{v_2},\ \underbrace{f_1 0\star\star}_{v_1}\underbrace{f_1 1\star\star}_{v_2},\ \underbrace{f_2 0\star}_{v_1}\underbrace{f_2 1\star}_{v_2},\ \underbrace{f_3\star 0}_{v_1}\underbrace{f_2 1\star}_{v_2},\}$$

where f_i is a factor of size i, or even

$$\underbrace{\star\star\star\star}_{v_1}\ \underbrace{\star\star\star\star}_{v_2} \setminus \{\underbrace{f_p 1 f_{s_1}}_{v_1}\underbrace{f_p 0 f_{s_2}}_{v_2}\}$$

where a f is a factor of any size.

The complexity of the computation of these predicates grows exponentially in the number of (bits across) cubes, i.e. variables, in their instances. The goal of our optimizations is then to minimize both the number of variables and sizes of cubes in any given program.

The clause specialization introduced in Chapter 9 reduces the number of variables occuring in primitive predicates, but also specializes the head of rules that, *in fine*, depend on facts from the EDB. This allows and fosters the use of the predicate specialization introduced in Chapter 11, which reduces the sizes of the cubes used in NoD.

An intuition of the effect of this transformation in our setting, network verification, is that it unrolls the topology and replaces predicates on the global states of all the network elements by local predicates on the state of, for example, a given switch. Then, the state of the ports or the packets received by the other switches will not be considered to compute the output of the switch. This processing *emulates* the style found in the examples provided with Network Optimized Datalog, such as shown in Section 5.2.1 or found in [Lopes,].

Example 12.1. Applying the partial instance and the predicate specialization to the program of Figure 6.4 transforms the definition of `linked(X, Y, IP)` into a set of specialized predicates `linked_X_Y(IP)` for each pair of linked locations `X` and `Y`. These new predicates are then described independently of the rest of the topology.

Remark 12.2. The analysis and optimizations introduced in this thesis have been designed with Network Optimized Datalog and Octant in mind, but their use is not limited to this context. In particular, we emphasize that the Coq formalization and proof of these tools are completely independent of the considered Datalog engine – actually, DatalogCert is a Datalog engine itself.

12.2 Towards a stronger static analysis

The analysis introduced in Chapter 10 overapproximates the behavior of a variable. To instantiate multiple variables in a single rule, the best we can do so far is to use a cross-product of the results of the different analyses (see Remark 10.21). The section sketches out a more precise but unsafe value extraction mechanism for such cases. Section 12.2.1 first makes an observation about the dependencies across value flows in Datalog, and how ignoring them entirely may lead to efficiency issues. We then relate it in Section 12.2.2 to the analysis as it has been formalized. Finally, Section 12.2.3 presents the limitations of this approach.

12.2.1 Minding dependencies across values

In Section 10.2.1, we identified three constraints of the T_P operator. The first two, conjunction and disjunction of candidate values, are enforced by the static analysis we introduced. However, some fundamentally different programs can only be distinguished using the third constraint, i.e. the unification of value tuples across atoms.

Example 12.3. Figure 12.1 shows two different definitions of a predicate p. In the program of Figure 12.1a, the number of deduced facts about p will be the same as the number of facts about q, whereas in Figure 12.1b, the former will be (up to) quadratic in the latter. However, the static analysis and extraction mechanism, as introduced in Chapter 10, are not able to distinguish both cases.

$$p(X, Y) \leftarrow q(X, Y)\,.$$

(a) Linear definition of p

$$p(X, Y) \leftarrow q(X, Z_1),\quad q(Z_2, Y)\,.$$

(b) Quadratic definition of p

Figure 12.1: Two definitions of p

Concretely, both definitions would be treated as the quadratic one, meaning that a program containing a similar construction could be instantiated using many irrelevant rules, which would hurt performances (Section 12.1 discusses this point in the context of our use case, Octant). Moreover, this dichotomy between dependent or independent values can be harder to determine, as illustrated by Figure 12.2, where the dependency between the values of `s` is lost at a deeper level than those of `q` were in Figure 12.1b.

$$\begin{aligned} &p(X_1, Y_1) \leftarrow q(X_1, Y_1)\,.\\ &q(X_2, Y_2) \leftarrow r(X_2, Z),\quad r(Z', Y_2)\\ &r(X_3, Y_3) \leftarrow s(X_3, Y_3) \end{aligned}$$

Figure 12.2: Deep quadratic definition of p

We can add an intermediate step, between the actual analyses and the extraction of values, that tries to take into account such dependencies.

12.2.2 Overlapping for better precision

This is where the annotations in the trees produced by the analysis come into play. The idea is to overlap the trees resulting from the analyses of multiple variables in a single clause, using the annotations to exclude incompatible flows, e.g. an atom being instantiated with different rules defining the corresponding predicate.

Example 12.4. In Example 10.9, variables X_1 and Y_1 appear in the same clause. The roots of their respective analyses (Figures 10.4b and 10.4c) have only one descendant, annotated with 0 in both cases. This implies that the two trees represent deductions that go through the same atom, i.e. p in the first rule. Then, we have $\vee$-nodes, and two descendants, annotated with 0 and 1. The left (resp. right) subtree represents in both cases a use of the first (resp. second) clause.

Mixing the values extracted from the left branch of one of the trees and the right branch of the other tree would amount to consider a program execution where the p atom in the body of the first clause is instantiated with both the first and second clause at the same time. This possibly can then be excluded.

Figure 12.3 shows what the overlapping of the two analyses using annotations would look like. The leaves are now pairs of value origins, as the idea of merging trees is to extract from the EDB tuples of values rather than individual values. The tree itself states that the values for X_1 and Y_1 can be extracted from the q facts of the EDB, either directly (left branch) or after permutation (right branch). More formally, X_1 and Y_1 can be instantiated using the values in $\{\langle x, y\rangle \mid q(x, y) \in EDB \vee q(y, x) \in EDB\}$, which matches exactly the actual behavior of the program.

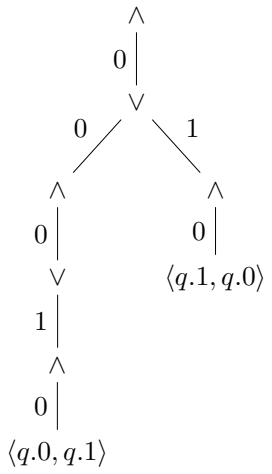


Figure 12.3: Merged analyses

This example shows how the annotations in the trees produced by the analysis can be used to avoid extracting many irrelevant values, where a quadratic – or worse, depending on the number of instantiated variables in a single rule – number of rules can be circumscribed to a linear process. However, although engaging, this process is not entirely safe.

12.2.3 Limitations

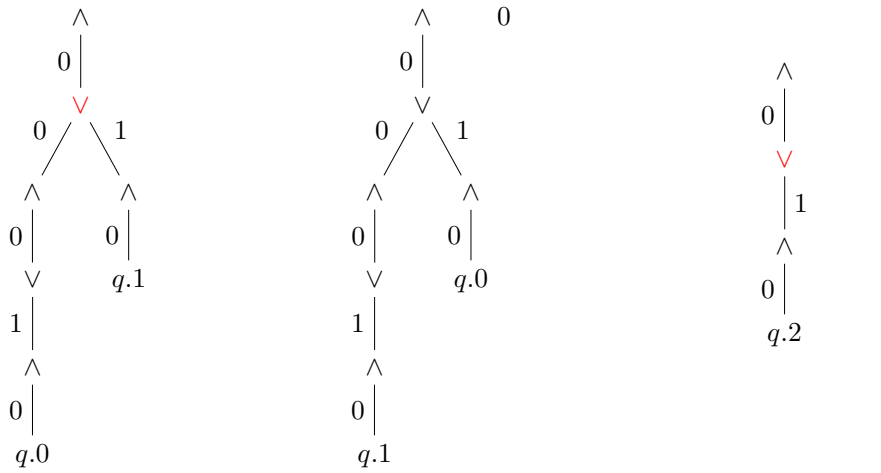
When applied to some programs, this limited extraction will drop relevant value tuples, as illustrated by the following example.

$$p(X_1, Y_1, Z_1) \leftarrow p(Y_1, X_1, Z_1)\,.$$
$$p(X_2, Y_2, Z_2) \leftarrow q(X_2, Y_2, Z_2)\,.$$

Figure 12.4: Mixing some values

Example 12.5. Consider the program in Figure 12.4. The analysis returns the trees shown in Figure 12.5. Let us now focus on the analyses of X_1 and Z_1, i.e. Figures 12.5a and 12.5c. If we try to merge them, things do not go as smoothly as before. Indeed, the ∨-node of the

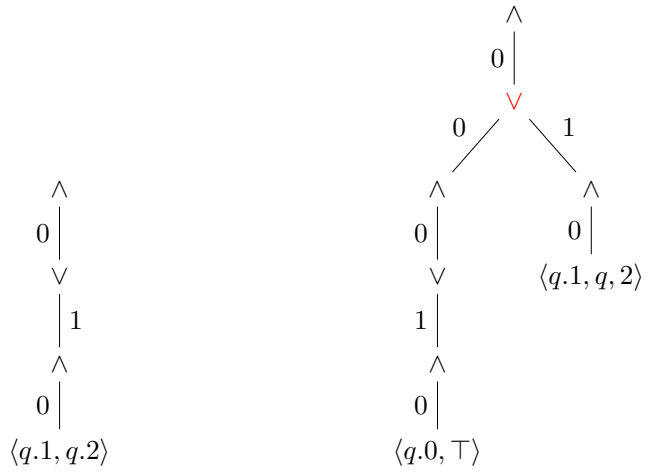
first tree, just under the root, has one more child than its equivalent node in the tree of Z_1. There are two ways to approach this: either drop the branch (indexed by 0) that has no counterpart, or consider it in the merger with a wild card $\top$ indicating an absence of related values. Figure 12.6 illustrates these two options.



(a) Analysis of X_1 **(b)** Analysis of Y_1 **(c)** Analysis of Z_1

Figure 12.5: Analysis of a heterogeneously recursive program

In Figure 12.6a, we rule out the possibility for a value to flow from $q.0$ to X_1, although it is clearly part of the semantics of the analyzed program, meaning that this method loses the completeness of the analysis. On the other hand, Figure 12.6b does not provide any more precision compared to what the *naive* process of Remark 10.21 would, as the $\top$ wild card should be replaced by the analysis of Z_1. In other words, this method, in this case, would only provide a circumvoluted unfolding of the normal one. Moreover, even though it is harmless in this example, there is currently no evidence that it would retain completeness when used on any program.



(a) Dropping the lonely branch **(b)** Keeping the lonely branch

Figure 12.6: Attempts at a difficult merger

This has not been explored yet, as we chose to focus on the *branch-dropping* path. Indeed, the program of Figure 12.4 is highly artificial, and actual Datalog programs, in our experience, do not contain this kind of mechanism. They rather tend to simply carry around full or partial value tuples without such local, partial permutations. In particular, this is exactly what Octant does, as illustrated by Figure 6.4.

Moreover, a preliminary implementation and certification of the analysis partially validated this form of the extraction. We indeed first designed an analysis, introduced in Section 12.3.4, that only considered non-recursive programs. In that setting, the no-recursion trace introduced in Section 10.4 was unnecessary, as we could prove that any trace directly matched a subtree of any analysis, in the sense that it could be *properly* overlapped. This way, the completeness results on traces could be *imported* in the certification of the mix of analyses.

When analyzing multiple variables in a given clause, the returned trees then all contained a subtree that fully matched the same trace. We were able to prove that there is no inconsistency between trees that match the same trace, thus ensuring that excluding incompatible branches did not break the completeness property, which provides a certification of this process in the context of non-recursive programs.

Recursion happens to be a core feature of Datalog, meaning that this result is encouraging but not satisfactory. However, we expect that there is a class of Datalog programs, strictly larger than non-recursive ones, where recursion is only used in a way that allows this more precise version of our analysis.

Our first intuition is that a program is *homogeneously recursive* if, for any given rule, all *argument cycles* have the same length. The idea behind this notion, which we do not define formally, is to see a rule as a graph whose nodes are the different atom arguments, and the edges relate head variables to their body occurrences, as well as the body arguments to the variables at the same index in the head.

Example 12.6. Figure 12.7 shows the *argument cycle* of variable X_1 in the head of the first rule of Figure 12.4. There is one occurrence of X_1 in the body, so we add an edge ($\rightarrow$) between these two occurrences.

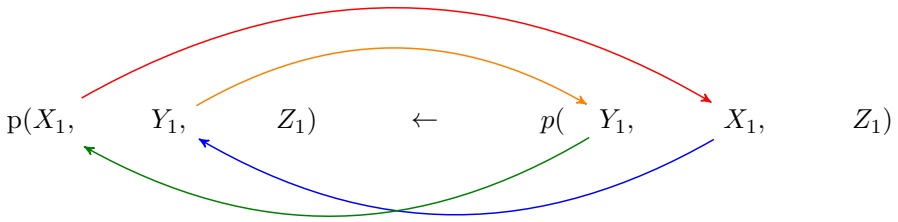


Figure 12.7: *Argument cycle* of X_1

This body occurrence is the second argument of the p atom, so we add an edge ($\leftarrow$) between the X_1 in the body and the Y_1 in the head. This head occurrence then leads us to the Y_1 in the body ($\rightarrow$), which gets us back to the X_1 in the head ($\leftarrow$). The result is a cycle of length 4. The same figure also shows that the cycle of Y_1 is of size 4 as well.

On the other hand, the argument cycle of Z_1 is of length 2, as shown by Figure 12.8. Our intuition is that the discrepency between the sizes of these argument cycles characterizes –

and possibly explains – the fact that the new extraction introduced in this section is not appropriate for the program of Figure 12.4.

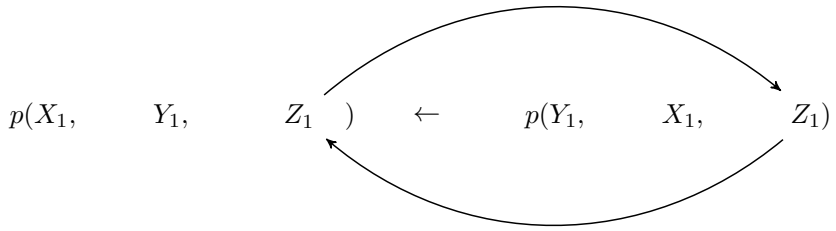


Figure 12.8: *Argument cycle* of Z_1

The rule studied in Example 12.6 contains only one head and one body occurrences for every variable. The next examples outlines how this notion deals with more occurrences.

Example 12.7. Let us consider the program of Example 10.10, which is replicated in Figure 12.9 for clarity. Like the program of Figure 12.4, it mixes values, but in a slightly different way.

$p(X_1, Y_1, Y_1) \leftarrow p(Y_1, X_1, X_1)\,.$
$p(X_2, Y_2, Z_2) \leftarrow q(X_2, Y_2, Z_2)\,.$

Figure 12.9: Shifting variables

Figure 12.10 shows the argument cycles of the rule. We start with X_1. Since it has two occurrences in the body, we add two transitions ($\rightarrow$) from the head occurrence of X_1. The other way around, i.e. from body to head, there can never be two edges from the same node, since these edges map a variable to the corresponding index in the head.

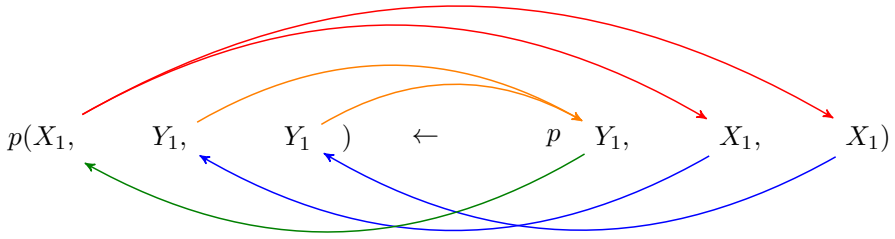


Figure 12.10: *Argument cycles* with multiple occurrences of a variable

We are left with two cycles of length 4. And, true to our intuition, the mix of the analyses of variables X_1 and Y_1 – which can be found in Figure 10.6 – works well. The result is shown in Figure 12.11, with the occurrences replaced by the corresponding predicate and index for clarity.

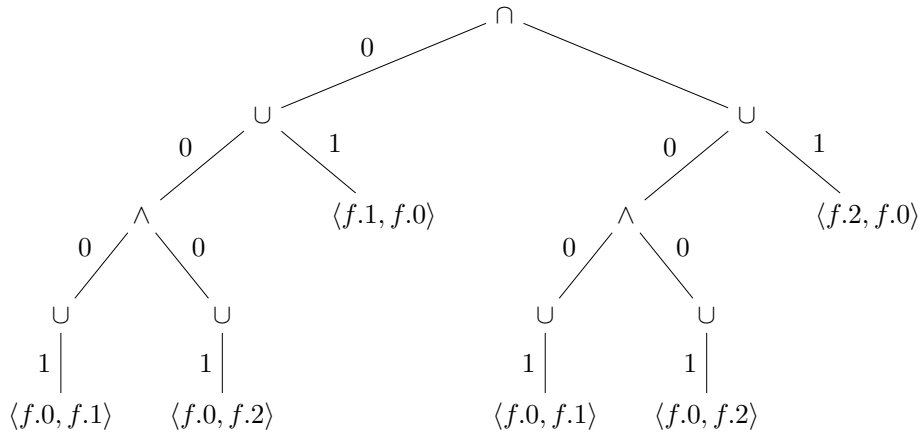


Figure 12.11: Mix of the analyses of X_1 and Y_1

These examples strengthen our intuition that our alternative, more precise extraction mechanism can be safely used on Datalog programs where all *argument cycles* within each rule are equal. However, this remains to be formally verified, in a proof assistant of course, but also on paper, as we do not even provide a core argument.

12.3 Modelization choices

Many modelization choices were made in the course of this work, both at the design and certification levels. This section discusses some of these choices, which have been more complex, surprising or critical than one could expect.

12.3.1 Implementation of pset

Figure 8.7 introduces the specification of a function called `pset`, which takes a set over an option type, e.g. `option A`, and filters out all the `None` elements to return a set over `A`. We provide two implementations of this `pset`, both of which leave us unsatisfied.

Figure 12.12 introduces a definition that roughly translates the given set into a list, which allows the use of the `pmap` function found in MathComp[1], which is very simply defined by induction on the list argument – a technique that, to the best of our knowledge, can not be used in the context of sets. The result in then translated back into a set.

```
Definition pimset {A B : finType} (f : A -> option B)
                  (s : {set A}) : {set B} :=
  [set id x | x in (pmap f [seq y | y in s])].

Definition pset {A : finType} (s : {set (option A)}) : {set A} :=
  pimset id s.
```

Figure 12.12: Defining pset using pmap

[1] https://github.com/math-comp/math-comp/blob/master/mathcomp/ssreflect/seq.v#L2626

We reckon that this implementation is rather inelegant, and somewhat reminiscent of what can be seen in some imperative programming languages. We introduce another formulation, which does not rely on another function. The idea is simply to go through every element of the provided set and only keep the *inner elements* of `Some` objects.

However, this approach requires us to name these *inner elements*. The top of Figure 12.13 shows the notation we wish we could use, which is not handled by Coq and MathComp. The bottom of the same figure displays the actual definition we use, where the *inner element* is universally quantified in the set definition.

```
(* Not accepted by Coq/MathComp *)
Definition pset_alt_target {A : finType}
                           (s : {set option A}) : {set A} :=
  [set y | x in s & x is Some y].

(* Used in practice *)
Definition pset_alt {A : finType} (s : {set option A}) : {set A} :=
  [set y | x in s, y in A & x == Some y].
```

Figure 12.13: Defining pset using set notations

The introduction of this quantification feels like a cheap trick, which moreover may impact the extraction of the function very inefficient in practice if the process is too naive. we do not know enough about the inner workings of MathComp to comment further on this precise point, but wanted to mention how surprised we were by the discrepency between the straightforward definition of `pmap`, and the fact that we could not find a simple and satisfying adaptation to the set type.

12.3.2 Introducing new (specialized) predicates

Chapter 11 introduces an optimization that rewrites a program where the ind^{th} argument of an intensional predicate f is always defined. The rewriting introduces new specialized predicates f_c for each constant c that appears at the ind^{th} index of the occurrences of f as the head of a rule.

To use these new specialized predicates in Coq, we assume a function of type `constant` → `symtype` (see Figure 11.5) and some associated hypotheses (Figure 11.6). Behind this approach, there are two Coq modelization choices we would like to discuss.

12.3.2.1 Where are the new symbols?

First, this method comes with an implicit, which is the presence *a priori* of those specialized predicates in the type of predicate symbols, `symtype`. DatalogCert [Benzaken et al., 2017b] defines `symtype` as a `finType`, and justifies it by recalling that the underlying model of **a** Datalog program is finite (see Section 8.1.1 of [Dumbrava, 2016]). More concretely, `symtype` is seen as a component of the program signature (Section 8.2.1 of [Dumbrava, 2016]), meaning that it corresponds to the set of predicates which appear in a given program.

In that setting, assuming that the set of f_c predicate symbols, which do not appear in the original program (see Figure 11.6), is in `symtype` is slightly contradictory. A more natural,

less tricky way to introduce those new predicates would have been to define a new `finType`, e.g. `spec_preds`, and define the signature of the new program as the union of `symtype` and `spec_preds`.

However, the atoms and ground atoms are defined *strictly* using `symtype`, meaning that this approach would have required to rework a lot of definitions. It may have been manageable if DatalogCert had split the predicate symbols into a type for the symbols themselves, and a set of such objects in the program signatures. This would nonetheless raise another problem, which is the definition of the type for predicate symbols not with respect to a given program. In that setting, it seems that there is no way or criteria to bound these symbols and obtain the `finType` required to define a `finset` over it.

In conclusion, assuming that the specialized predicate symbols are in `symtype` is not fully satisfactory, but it would seem that there is no simple and cleaner alternative to circumvent an issue that is specific to the Coq formalization.

12.3.2.2 When genericity leads to troubles (bis)

The new predicates are introduced *via* the `pclone` function (see Figure 11.5), whose only argument is a constant, because it is implicitely applied to function f at index ind. It might have seemed more natural and general to have a specialization function that can be applied to any predicate symbol and argument index, but this may have led to a paradox in the hypotheses, or a least some needless troubles.

As explained just above, in Section 12.3.2.1, the new specialized predicates are packaged with the original ones in `symtype`. Having a more general version of `pclone` would then have meant that it could have been applied recursively to the new predicates. This would have not been compatible with the hypotheses we introduce (see Figure 11.6). In particular, hypothesis `parity` (on the arity of these new predicates) implies that a predicate that has been specialized until it reaches the absence of arguments should stay at this *level* when the specialization is applied again.

However, we also need the specialization to be injective (cf. hypothesis `pinj` in Figure 11.6), meaning that we could not simply map an empty predicate $P()$ to itself. A workaround would be to create a hierarchy of predicates $P() \mapsto P_1() \mapsto P_2()$..., but that would contradict the finiteness of `symtype`. A more valid but less natural fix would be to have a *circle* of benign specializations, such as $P() \mapsto Q() \mapsto P()$. This method illustrates how, in the setting of a generalized `pclone`, paradoxes may be avoided, but still lead to convoluted, abstruse problems. On the other hand, limiting the use of the specialization to a given predicate (f) at a given index (ind) makes these very questions irrelevant.

This is in constrast with our general experience with Coq, where introducing genericity may lead to much more flexibility and convenience, in particular for the use of inductions. Such an example is also found in the definition and certification of the predicate specialization. See for example the Coq snippets of Section 11.3.3), where the `j` argument is fundamentally a stand-in for `ind` that allows proofs *via* induction.

12.3.3 Bound of the traces

The proof introduced by Remark 8.14 basically states that having a repetition of a clause and substitution couple in a trace would amount to proving a fact x as an intermediate step in the

proof of x, which we can do without. However, different couples of clause and substitution can represent the same (head) fact, meaning that even with the `WUtree` type, a same fact can be deduced multiple times in a single trace.

Although this *bounded* form of repetition is harmless, as it is already finite and functional, it may feel a bit awkward and unsatisfying. We then considered the possibility to define a type of trees with unicity across branches *modulo* a function (here, $\lambda C.\lambda\iota.head(\iota(C))$). However, we dropped this idea due to its lack of *actual benefit*, but would like to highlight that it would change (and lower even further) the surprising bound discussed in Section 10.4.4.

12.3.4 Implementation of the static analysis

Section 6 of [Tristan and Leroy, 2008] states that "generally speaking, there are two ways to specify an algorithm in Coq: either as inductive predicates using inference rules, or as computable functions defined by recursion and pattern-matching over tree-shaped data structures". Even if Datalog programs are not naturally seen as trees, both from syntax[2] and semantics standpoints, the version of the static analysis introduced in Chapter 10, is a computable function.

However, this development is the result of many reworks and adjustements, and used to be defined using an inductive. We outline this preliminary version, and explain why we departed from it.

12.3.4.1 Original paper definition

In our first approach, the analysis was defined on paper in the form of a typing system, and already returned trees encoded by Disjuncted Normal Forms. The types are built using the following rules, which preserve the DNF.

- $\top$ is a wildcard type, which provides no information on the analyzed variable or predicate argument. It was introduced in this version to modulate the thoroughness of the analysis.

- $tInit \stackrel{\text{def}}{=} \{\{[::]\}\}$, the base type for extensional predicates.

- $tInsert : tocc \to Types \to Types$ adds a $tocc$ at the top of each path under the root. Given $tocc\ a$ and $Disj\ d$, it returns

$$\{\{a :: b \mid b \in c\} \mid c \in d\}$$

  If the input type is $\top$, then $\top$ is also returned.

- $tDisj : Types \to Types \to Types$ is simply set union, corresponding to the aggregation of possible sources of values. If one of the input types is $\top$, then $\top$ is returned as well.

[2] Although they implement them as lists in the formalization, Datalog programs are even defined as sets of clauses in [Dumbrava, 2016].

- $tConj : Types \to Types \to Types$. Given input $Disj$s d_1 and d_2, it returns

$$\{x \cup y \mid x \in d_1 \ \wedge \ y \in d_2\}$$

  This operation is equivalent to putting d_1 and d_2 under a $Conj$, while preserving the DNF. If one of the input types is $\top$, the other type is returned.

These constructs are then used by the typing rules shown in Figure 12.14, where P is the analyzed program and the list of toccs already visited (`prev` in Figure 10.18) is formalized as the typing context Γ.

$$\frac{\text{p is an extensional predicate}}{\Gamma \vdash p.i : tInit}\ \textbf{predb}$$

$$\frac{\forall (C : p(\overrightarrow{v}) \leftarrow \cdots) \in P,\ \Gamma \vdash v_i : \tau_C \qquad \text{p intensional}}{\Gamma \vdash p.i : \underset{(C:p(\overrightarrow{v})\leftarrow\cdots)\in P}{tDisj}\ \tau_C\ \emptyset}\ \textbf{predr}$$

$$\frac{\forall \langle x,y,z\rangle \ \in \ occs(v)\backslash\Gamma,\ \Gamma \cup \{\langle x,y,z\rangle\} \vdash (p\_at(\langle x,y,z\rangle)).z : \tau_{\langle x,y,z\rangle}}{\Gamma \vdash v : \underset{\langle x,y,z\rangle\in occs(v)\backslash\Gamma}{tConj}\ (tInsert\ \langle x,y,z\rangle\ \tau_{\langle x,y,z\rangle})\ \top}\ \textbf{var}$$

Figure 12.14: Core typing rules

We do not detail these rules, but they should be reminiscent of the algorithm introduced in Section 10.2, although presented in a circumvoluted (and somewhat inelegant) manner.

12.3.4.2 Original Coq implementation

Figure 12.15 shows the Coq formalization of these typing rules. They are encoded as four mutually-defined inductives, which are supposed to emulate a loop-based implementation. We do not dwelve into this code, or introduce every definition it relies on, as a simple look at it should be enough for a comparison with the implementation shown in Chapter 10.

12.3.4.3 From Inductive to sets

We have been working on and with this version of the static analysis for approximately a year and a half. During this interval, there were some technical difficulties, mainly the fact that the induction principle required four manually-defined invariants and produced many, sometimes abstruse proof obligations. Moreover, this presentation did not allow us to reason about the termination of the analysis, which we felt was really missing. As a corollary, there was also something awkward about defining what is supposed to be a deterministic function as an inductive predicate.

A posteriori, we reckon that these elements would and should have been reason enough for us to erase the inductive definition of the analysis and start working on another version. However, the reason that changed our mind was realizing that the inductive definition is actually flawed: whenever it is used on a recursive program, it returns the trivial type $\top$.

```
Inductive predTyping : forall p v (ctxt : (tocs p)) (f : symtype),
                           'I_max_ar -> (DDtype ctxt) -> Prop :=
  | pt_base : forall p v (ct : tocs p) f j,
                (predtype f = Edb) ->
                @predTyping p v ct f j (mk_DDtype (@tInitDDtype p ct))
  | pt_rec : forall p v (ct : tocs p) pred (j : 'I_max_ar) typs,
              predtype pred = Idb ->
              @progPredTyping p v ct p pred j typs ->
              @predTyping p v ct pred j (fold_type_alg DtDisj typs DEmpty)
with
 varTyping : forall p (ctxt : tocs p), 'I_n -> (DDtype ctxt) -> Prop :=
  | vt : forall p (ct : tocs p) (tot : 'I_(bn*max_ar).+1) v occsTypes,
                              (* getting rid of stuff that has already been typed *)
      @OccsToTypes p v ct (seq_to_enotin (occsInProgram p v) ct) occsTypes ->
      @varTyping p ct v (fold_type_alg DtConj occsTypes (mk_DDtype (@TrivDDtype p ct)))
with
  (*  full prog -> context -> intermediate prog -> pred -> ind -> intermediate types *)
  progPredTyping : forall p v (ctxt : tocs p) (ip : program)
                          (f : symtype) (ind : 'I_max_ar),
                          seq (DDtype ctxt) -> Prop :=
  | ppt_base :    forall fp v ct pred j, @progPredTyping fp v ct [::] pred j [::]
  | ppt_rec_no :  forall p v (ctxt : tocs p) ip pred new_cl j typs,
                  @progPredTyping p v ctxt ip pred j typs ->
                  (pred <> (hsym_cl new_cl)) ->
                  @progPredTyping p v ctxt (new_cl :: ip) pred j typs
  | ppt_rec_yes : forall p v (ctxt : tocs p) ip new_cl j typs ntyp v',
                  @progPredTyping p v ctxt ip (hsym_cl new_cl) j typs ->
                  (nth_error (arg_atom (head_cl new_cl)) j) == Some (Var v') ->
                  @varTyping p ctxt v' ntyp ->
                  @progPredTyping p v ctxt (new_cl :: ip)
                                  (hsym_cl new_cl) j (ntyp :: typs)
with
  OccsToTypes : forall p (v : 'I_n) (ctxt : tocs p), {set (enotin ctxt)}
                       -> seq (DDtype ctxt) -> Prop :=
  | colt_base : forall p v (ct : tocs p),
        @OccsToTypes p v ct (seq_to_enotin set0 ct) [::]
    (* The set l has (recursively) been typed as the list typs. Adding occurrence
       tocc to the set triggers a call to predTyping, with tocc added to the context *)
  | colt_rec :  forall p v ct tocc l typs
                       (dt : DDtype (ct :|: [set (elnotin tocc)]))
                       pato rul_ind body_ind aind,
                @OccsToTypes p v ct l typs ->
                (elnotin tocc) = (T_occ rul_ind body_ind aind) ->
                p_at (val tocc) = Some pato ->
                @predTyping p v (ct :|: [set (elnotin tocc)]) pato aind dt ->
                @OccsToTypes p v ct ([set tocc] :|: l)
                             ((@DtInsert p ct (elnotin tocc) dt (Helnotin tocc)
                             (@ddtextract p ct (val tocc) dt))::typs).

Scheme varTyping_mrec := Induction for varTyping Sort Prop
  with occsToTypes_mrec := Induction for OccsToTypes Sort Prop
  with predTyping_mrec := Induction for predTyping Sort Prop
  with progPredTyping_mrec := Induction for progPredTyping Sort Prop.
```

Figure 12.15: Original Coq implementation of the static analysis

This *bug* comes from the use of $\top$ as the "base case" of the fold of $tConj$ in rule `var`, when $occs(v)\backslash\Gamma$ is empty, which *infects* the whole return type through $tDisj$.

The point of certification is of course to avoid this kind of subtle and technical but critical error. We had written and proved a completeness lemma, roughly stated as "if the analysis returns an actual result (or type), it does capture an overapproximation of the behavior of the analyzed variable". We were sure – and wrongfully so – that the analysis would always return a non-trivial result, so we thought we could safely add the hypothesis on the returned type to help with the technicalities of the proof, which in fact were *legit* safeguards.

We eventually understood that something was wrong thanks to our work on the smarter extraction mechanism introduced in Section 12.2. Concretely, after formalizing it in Coq, we could certify it with a lemma roughly stating that "any trace directly matched a subtree of any analysis, in the sense that it could be *properly* overlapped. This way, the completeness results on traces could be *imported* in the certification of the mix of analyses" (cf. Section 12.2.3).

Both the traces and analyses are trees bounded in height, but by different values (number of rules times the cardinal of the substitution type for the traces, number of *tocc*s for the analyses). In particular, the bound for the analysis is tighter, meaning that our lemma probably contained a contradiction.

We investigated the problem by reflecting on the proofs, and noticed that Datalog recursion was actually not dealt with in the certification of the inductive version of the static analysis. One of the reasons we had not realized that when actually writing the proofs is quantity and very technical – sometimes obscure – nature of the proof obligations generated by the four mutually-defined inductives. In comparison, working with the set-based version was much clearer and allowed easier high-level reasoning.

We eventually switched to the set-based definition seen in Figure 10.18. Since we had no experience with MathComp prior to this project, and could not even find a satisfactory paper definition of the analysis, we first hoped little of the Coq version. MathComp finTypes and set notations had already been noted particularly relevant to formalize Datalog [Benzaken et al., 2017a], and it was as well for our static analysis. Although the definition is not completely straightforward, the intricacy seems inherent to the analysis rather than a consequence of the formalization itself.

The authors of [Tristan and Leroy, 2008] also recall that defining a function, such as our analysis, in a computational way rather than as an inductive also allows its extraction as an Ocaml program (which we have not experimented with this development yet). Alternatively, numerous non-functional programming languages (e.g. Python) now support set notations. Another advantage of this version of the analysis is then its simplified translation in many languages, which reduces the gap between formalization and implementation, making the latter more trustworthy.

In summary, even in the context of machine-aided verification, the mix of an error in a minor definition – which would have been spotted with correct lemmas – and lacking formulations of completeness properties – which would have been benign with correct definitions – could lead to a broken result. Computational definitions, higher-level tools (both provided by MathComp in our case), not adding "apparently free hypotheses which help with the technicalities of proofs", and a more introspective view should help avoid this kind of situation.

12.4 General proof effort

Although the use of finite types and set notations was eventually most beneficial to us, our proof style remained more classical. This is in contrast to [Benzaken et al., 2017b], which uses the tactic language SSReflect extensively. Combined with the heavy use of dependent types to obtain finite types, it resulted in a development that was probably longer than what could be expected, i.e. approx. 7000 lines of code. In comparison, the positive Datalog engine within DatalogCert we use consists of approx. 1500 lines of code.

Our development can be found at `https://orange-opensource.github.io/octant-proof/`, and its 7000 lines of code are roughly split as follows:

- 1000 loc. for a general-purporse library used throughout our development (file `utils.v`)
- 400 and 1200 loc. for the finite sequence and tree types introduced in Chapter 7 (files `finseqs.v` and `fintrees.v`)
- 1000 loc. for the design and certification of the Datalog trace semantics introduced in Chapter 8
- 300 loc. for the definition and certified collection of variable occurrences, as presented in Section 10.3.2 (file `occurrences.v`)
- 300 loc. for the design and certification of the partial program instance introduced in Chapter 9 (file `rinstance.v`)
- 450 loc. for the no-recursion traces introduced in Section 10.2.2 (file `norec_sem.v`)
- 450 loc. for the design and high-level certification (many technical results are found in other files) of the static analysis introduced in Chapter 10 (file `static.v`)
- 200 loc. for the extraction of values from a static analysis (file `extract_static.v`)
- 1500 loc. for the design and certification of the predicate specialization introduced in Chapter 11 (file `projection.v`)
- 300 loc. for a preliminary implementation of the alternative value extraction mechanism introduced in Section 12.2 (file `dep.v`)

  Finally, some additions to DatalogCert have been made *in situ*, and are flagged with `-- added`.

12.5 Related works

The work presented in this thesis is at the interface of various domains, which can roughly be abstracted as network verification, logic programming, and (certified) program analysis and transformation. The network verification component is fundamentally a background that explains how and in what context we started this work, so we introduced it earlier, in Part III. On the other hand, this section discusses research works that are related to the concrete questions we addressed and the answers we provided.

12.5.1 Static analysis for logic programs

The vast majority of static analyses for logic programming languages are developed for Prolog. For example, [Jacobs and Langen, 1992, Marriott et al., 1994] introduces a general abstract framework for the static analysis of Prolog programs, and provides an example that focuses on the groundness and sharing of variables across terms to set up parallelism in the program's execution.

Prolog was extended into λProlog [Nadathur and Miller, 1988], which contains a typing system. As a side note, the static analysis we introduce could very easily be leveraged to determine the type of variables in a Datalog program, as it relates those variables to values in the EDB which are available at compile time, although it probably would not be the most efficient typing method.

Such typing systems for Datalog can be found in [Zook et al., 2009] and [de Moor et al., 2008]. The latter also introduces two type-based optimizations: type erasure, which removes dynamic type tests once a program has been shown to be type-safe, and type specialization, which specializes predicate definitions to the type contexts in which they are called, e.g. removing clauses which are shown to contain type inconsistencies.

There has been some work [Chaudhuri, 1993, Chaudhuri and Kolaitis, 1994] that aims at approximating potentially recursive predicates *via* a set of nonrecursive, simpler rules. However, this line of research has not produced many usable results, as many classes of predicates, such as transitivity closure, are not approximated in a satisfying manner by these methods [Duschka, 1998].

The domain of a program can be leveraged by static analyses and transformations. For example, [Campagna et al., 2011] introduces a source-to-source transformation of Datalog programs handling arithmetics, that relies on propagation techniques from constraint programming.

The work introduced in [Miller, 2006, Miller, 2008] views Horn clauses as instances of a higher-order logic (e.g. Simple Theory of Types), which makes it possible to see the execution of programs as cut-free sequent calculus proofs, which in turn allows the presentation of these executions through the lens of linear logic. λProlog is then used to implement a successful prototype tool that approximates the content of multiple data-structures (lists, sets...) within logic programs.

The authors of [Mesnard and Neumerkel, 2001] introduce a static analysis that tries to infer the termination of Prolog programs, using an abstraction of the terms by their height and the computation of numerical relations between the values in rules. The vigilant reader may notice that Definition 3 of this paper, which describes the cornerstone of their analysis, is reminiscent of the one we introduce in Figure 8.9. In a sense, we both try to capture the variable-level logical structure of the analyzed program, although with a different approach and problem in mind (Datalog programs do not contain terms and terminate by construction).

The closest work to our own is probably found in [Halevy et al., 2001]. This paper introduces a static analysis of Datalog programs that represents as a (set of) tree(s) not only the dataflow of a variable, but the whole program. Another similarity is that they also limit the recursion of Datalog in their analysis to be able to encode in a finite manner an infinite (or rather *sufficient*, see Remark 8.14 of the present document) number of deductions, although with different criteria and implementation: they do not allow to have in a branch two rules whose heads have the same predicate and *variable pattern*.

This may sligthly echo the approach we propose in Remark 12.3.3, although their formalism does not account for the substitutions (or as an abstract and simplified form they call labels). This is due to the fact that their analyses do not try to convey the same information as ours, as they aim less at the actual behavior of the analyzed program than the decidability of properties such as satisfiability (existence of an EDB such that at least one fact about a predicate is decidable) or equivalence, for which they provide instances of their general method and a characterization of the relevant class of Datalog programs.

In the same spirit, the authors of [Caballero et al., 2008] introduce a debugging method for Datalog programs that focuses on semantics rather than actual computation mechanisms. Concretely, they introduce a structure called *computation graphs*, that finitely represents relations between predicates and is used to investigate discrepencies between the expected and actual semantics of a given program.

They oppose their approach to *computation trees*, which have for example been used for Prolog (SLD-trees, cf. [Lloyd, 1987a]) or Java [Caballero et al., 2007], claiming that tree-like structures are not fit to handle the always-terminating recursion of Datalog. We did introduce a trace semantics for Datalog in Chapter 8, but its aim was only to be able to reason about full deductions at the proof level, and not be used as an actual debugging tool. We however believe that our trace semantics can be used as a simple, off-the-shelf tool to investigate the deduction of an unexpected ground atom. On the other hand, unlike [Caballero et al., 2008], it would not help with the absence of an expected fact.

Soufflé [Scholz et al., 2016] is a static analysis tool using Datalog as a specification language. It performs Datalog-level optimizations, such as magic sets (specialization of a program w.r.t. a query [Balbin et al., 1991]) and user-directed rule inlinings. The Datalog code is translated into a relational algebra *via* the Futamura projection, where the interpreter is the semi-naive evaluation [Abiteboul et al., 1995]. Although a form of specialization, this transformation – not discussed or illustrated – does not seem to produce an explicit analysis of the program value flows.

Apart from the aforementioned papers, there has not been much research in the field of static analysis for Datalog. Ironically, it has recently been used as a framework to build static analysis tools, such as Soufflé, but also [Whaley et al., 2005, Arntzenius and Krishnaswami, 2016, Madsen et al., 2016]. The rationale is that Datalog, as a language, is close to the logic usually underlying the specification of a static analysis [Greenman, 2017].

12.5.2 Dealing with recursion for static analyses

The main difficulty we faced in this work is the development and certification of a finite representation for an infinite set of behaviors. This challenge is found in the development of the analysis (Section 10.2.2), the implementation of our trace semantics (see Remark 8.14), and the proof of their connection (Section 10.4.1). In all these cases, the name of the game is to truncate our representations enough to make them finite, while retaining the required information.

This general process is reminiscent of earlier works in the field of verification, such as [Shivers, 1991], which introduces the k-CFA family of static analyses. This work tackles the problem of dynamic dispatch for the design of static analyses for higher-order, functional languages, i.e. the fact that the flow of values has an impact on the call-target resolution procedure [Might et al., 2010]. The k integer parameter of k-CFA, called *context-sensitivity*, is the number of previously visited call sites which are considered. This truncated notion

of trace ensures finiteness. Note that this work *linearly abstracts* the traces, in the sense that the last k steps are all considered and the rest is left out, whereas we fundamentally ensure finiteness by allowing the analysis or trace semantics to drop multiple, not necessarliy contiguous set of steps.

As another example, [Jeannet and Serwe, 2004], introduces an *abstract semantics* for imperative programs that roughly abstracts the sequence of intermediate contexts (i.e. values associated to global variables) in a trace by any such sequence that would have led to the final context. Note that, in this case, the semantics is not truncated into a technically finite representation, but still a more manageable one.

12.5.3 Certified static analyses

David Pichardie and his colleagues work on the development and certification of static analyses within Coq [Cachera et al., 2004, Besson et al., 2006], which are then extracted using Letouzey's method [Letouzey, 2008]. The main application area of this work is the analysis of Java [Barthe et al., 2013] but is also notably used in the development of CompCert [Leroy, 2009, Barthe et al., 2017].

Chapter 3 presents a Coq formalization of Datalog, called DatalogCert. It is part of a lager project, called DataCert[3], which aims at building a fully and deeply verified environment for data intensive management tools, the same way CompCert [Leroy, 2009] and CakeML [Kumar et al., 2014] provide verified realistic C and ML compilers. As a side note, Kriener et al. used Coq in [Kriener et al., 2013] to prove the equivalence of different Prolog semantics.

As far as we know, our work is the first formally proved implementation of non-trivial static analyses and rewritings for a declarative and popular language, Datalog. It is also the first full-blown application of DatalogCert. Although we had to slightly extend the formalization, our work shows that it can concretely be used to prove concrete and non-trivial results on the use of Datalog, giving credits to the ambition of DataCert to provide a full environment for Datalog, among other aspects of data intensive applications.

12.5.4 Exploring Datalog subclasses

In Section 12.2, we introduce a stronger static analysis and lay out the hypothesis that there exists a subset of Datalog programs, strictly greater than the set of recursion-free programs, on which the analysis is complete. The idea of restricting Datalog to a viable or more efficient fragment is found in other works.

For example, [Reutter et al., 2015] identifies *Regular Datalog*, i.e. the class of programs using binary predicates with recursion limited to transitive closures, as a suitable Datalog fragment for graph queries. The Regular Datalog theory and an efficient, incremental engine for this fragment have then been implemented and certified in Coq [Bonifati et al., 2018]. Due to timing constraints, we did not investigate this side of research about Datalog, but such works would obviously be a natural and strong starting point for a precise characterization of a Datalog fragment that could handle our stronger static analysis.

[3] http://datacert.lri.fr/

Part VI
Conclusion

La route suivie jusque-là était comme un rêve dont la trace s'effaçait au fur à mesure qu'elle avançait

---

Hwang Sok-Yonh, *Shim Chong, fille vendue*,
traduit du coréen par Choi Mikyung et Jean-Noël Juttet

Summary and perspectives

The work presented in this document is, as it is often the case in research, the adventitious result of the proverbial *two-week side project*. More concretely, this thesis was supposed to be about the NetKat language (see Section 4.4), but we have been asked to take a quick look at the Network Optimized Datalog engine (Chapter 5) to help with the performances of Octant (Chapter 6). The result is, as outlined in the introduction, the identification of a caveat in the aforementioned engine, the design of a static analysis, two program transformations and a trace semantics for Datalog, their certification within an existing Coq/MathComp formalization of the language and the introduction of new finite types for MathComp.

We strongly believe that one of the key *features* of the present thesis is the fact that it leverages DatalogCert (see Chapter 3) and provides a nontrivial proof-of-concept and strong argument for its reusability and scalability (which are quickly discussed in the conclusion of [Dumbrava, 2016]). Moreover, understanding the design and implementation choices behind this engine, which led us to propose some additions or modifications, was a very interesting and enjoyable intellectual journey. It goes without saying that we are very grateful to the authors of DatalogCert – Ştefania-Gabriela Dumbravă, Véronique Benzaken and Evelyne Contejean –, without whom our work would clearly have lacked strong foundations and justifications.

On a more concrete level, the static analysis and transformations we introduce (usually) clear and short specifications into a network-specific form closer to NoD programs. Doing so by hand is obviously possible, but also lengthy, complex and error-prone, meaning that our certified optimizations conciliate performances and safety. The rewritten programs are computed orders of magnitude faster than the original ones, but remain significantly slower than their hand-written NoD counterparts in some instances. This justifies the work on a *smarter* static analysis (see Section 12.2), which tries to trade a bit more safety (it becomes inadequate for some Datalog programs) for much more efficiency. The key question is then to determine precisely what programs are excluded – in particular provide a simple syntactic criteria – and if the remaining programs are relevant. We outline answers to these questions, but they remain to be formalized and verified.

Moreover, we emphasize the fact that the static analysis and the two rewritings we introduce are all defined and certified independently. In particular, should someone find a better static analysis for Datalog than ours (Chapter 10) – may it be the alternative version mentioned just above or an entirely different approach –, it could very easily be *plugged* into the partial program instantiation of Chapter 9, notably thanks to the very broad definition of the completeness hypothesis on the provided set of substitutions (see Definition 9.4).

The other way around, the work presented in this document has been designed with Network Optimized Datalog and network verification in mind, but may be helpful in other contexts. This possiblity remains to be investigated.

Lessons learned

During the course of this thesis, we attended a summer school where Andrei Paskevich stated that "Roughly speaking, the formal certification of a system is an order of magnitude more complex and time-consuming than the actual design of said system". This result, which may sound abstract to some, has ended up feeling very real to us. In particular, we were baffled by how very simple and straightforward ideas, such as those behind the static analysis we introduce, can be implemented in intricate and error prone – sometimes misleading – ways, and require complex, layered reasonings to be certified.

This brings us back to the conclusion of [Benzaken et al., 2017a], which underlines that the justification of many foundational and "intuitively clear" database results had always been treated with a high-level persective rather than "scrupulous proofs", meaning that "low-level details were either glanced over or left to the reader". Knowing that there is a continuity and *coherence* in terms of such difficulties and motivations between the implementation of DatalogCert and our own work has been of great comfort to us.

The other core lesson (or rather set of lessons) we learned during this work is more qualitative than quantitative, and found in Section 12.3.4. This section basically explains that

1) our first implementation of the static analysis contained a bug

2) and was designed in a way that made the proofs much harder and less clear

3) which led us to write a deficient completeness statement that allowed the bug to go unnoticed for a long time.

We have been through many formal methods classes – and have been told roughly as many times about the crash of Ariane 5 Flight 501 –, so we were not entirely surprised by 1). This kind of "the devil is in the detail" situation is, after all, why we need formal methods.

The lesson of 2), like the statement of Andrei Paskevich discussed above, was already *theoretically known* but not concretely experienced by us. The choice of adequate models and formulations is of course discussed, or at least introduced, in formal methods classes, but the systems and algorithms which are verified in such settings are usually defined *a priori*. This means that we never had to work on the certification of an inadequately formulated component and did not expect this aspect to be as critical as it has been to us.

Finally, the issue that taught us the most was undoubtedly 3). We have been *raised* – in an academic sense – to think that the specification and verification of algorithm or program properties was an extremely powerful process. This intense promotion of certification may have led us to believe in the *all mightiness* of the existence of a verification pipeline, and thus not put enough care into its foundations, i.e. the statements that are checked. Throughout the rest of our life as a formal methods enthusiast and practitioner, we will remember this experience to help us reach and maintain a high level of self-criticism and perspective.

Appendix A

Coq basics

This document presents a research work that has been designed, developed and certified using the Coq proof assistant. As such, Parts IV and V contain a lot of Coq code, and assume some familiarity with it. In case this document would ever find its way into the hands or screen of anyone who has strictly no experience with Coq or a similar proof assistant, this appendix provides a very simple, high-level and usage-oriented introduction.

This introduction does not dwelve into the theoretical foundations of proof assistants, and it is not designed to replace a proper academic course[1]. It should, however, convey the *spirit* of Coq and help a confused reader get a grasp of what is going on in the aforementioned parts of this document, if not the full details.

Section A.1 introduces the interactive nature of Coq. Then, Section A.2 illustrates it with two simple proofs in first-order logic. Finally, Section A.3 discusses the use of Coq in the development of verified programs and compilers.

A.1 Interactive theorem proving

A proof assistant is, as the name suggests, a tool designed to help people prove results. To do so, the user must be able to define objects (logics, algebras, programs...) and state properties about them.

In the introduction of his PhD thesis [Winterhalter, 2020], Théo Winterhalter provides a great discussion on proof assistants, in which he compares them to chatbots. Apart from the fact that it is sometimes surprisingly hard and frustrating to make oneself understood by a proof assistant, the main similarity is the "feedback loop" working environment it sets up.

Once the user has formally defined an object and a statement he or she wants to prove, called the goal, the proof assistant awaits a step of the proof. Each time the user provides one, the proof assistant checks that it indeed can be applied, computes the new goal, i.e. what is now left to prove, and expects a new proof step. This process terminates once all these steps form a full proof of the initial statement.

[1] See https://wikimpri.dptinfo.ens-cachan.fr/doku.php?id=cours:c-2-7-1 for the foundations and https://wikimpri.dptinfo.ens-cachan.fr/doku.php?id=cours:c-2-7-2 for a comprehensive presentation of Coq as a tool.

Notation A.1. This document tries to reproduce the interface used by Coq during proofs. Figure A.1 displays the general shape. The left side shows a list of instructions, or proof steps, entered by the user. On the right is the list of current (i.e. after the execution of the list of instructions) hypotheses and the current goal, i.e. the formula that still needs to be proved to obtain the original statement.

```
instruction 1.
instruction 2.
instruction 3.
```

```
Hypothesis 1
Hypothesis 2
-----------------
Goal
```

Figure A.1: Feedback in Coq

This process and the associated notation are illustrated by the next Section, which introduces the base use of Coq, i.e. the manipulation of first-order logic.

A.2 Playing with first-order logic

We provide two detailed examples, introducing first some propositional-level proof, and then how first-order quantifiers are handled.

Example A.2. We start with a proof of the commutativity of conjunction in formal logic, which we state as $\forall A, \forall B, ((A \wedge B) \rightarrow (B \wedge A))$. Assuming for simplicity that A and B are already defined elsewhere as logical propositions, this lemma, which we name `and_comm`, is easily translated in Coq, as shown in Figure A.2.

```
Lemma and_comm: A /\ B -> B /\ A.
```

Figure A.2: Commutativity of $\wedge$ in Coq

Telling Coq that we want to prove this statement would trigger the display of Figure A.3.

```
A, B : Prop
-----------------
A /\ B -> B /\ A
```

Figure A.3: Starting a proof

So far, the list of proof steps is empty, and the only hypotheses are the existence of (abstract) propositions `A` and `B`. The goal is of course the statement of `and_comm`. Figure A.4 shows the first step: moving the `A /\ B` proposition from the goal to the hypotheses, naming it `H`.

```
intro H.
```

```
A, B : Prop
H : A /\ B
------------------
B /\ A
```

Figure A.4: Moving a precondition to the hypotheses

As shown, we now need to prove `B /\ A`, with the hypothesis `A /\ B`. To do so, we break down `H` into two smaller hypotheses `A` (named `HA`) and `B` (`HB`), as shown by Figure A.5.

```
intro H.
destruct H as [HA HB].
```

```
A, B : Prop
HA : A
HB : B
------------------
B /\ A
```

Figure A.5: Breaking down an hypothesis

Since the goal, `B /\ A` is two-fold, we (literally) split it into two subgoals in Figure A.6.

```
intro H.
destruct H as [HA HB].
split.
```

```
A, B : Prop
HA : A
HB : B
------------------ (1/2)
B
------------------ (2/2)
A
```

Figure A.6: Splitting a goal into subgoals

This opens two new branches in the proof, which we keep track of with indentation. Since the new subgoals directly correspond to one of the hypotheses, we simply apply them, as shown in Figures A.7 and A.8.

```
intro H.
destruct H as [HA HB].
split.
- apply HB.
```

```
A, B : Prop
HA : A
HB : B
------------------
A
```

Figure A.7: Applying an hypothesis matching the goal

```
intro H.
destruct H as [HA HB].
split.
- apply HB.
- apply HA.
```

```
No more subgoals.
```

Figure A.8: Ending the proof

This closes all open subproofs, and thus marks the end of the proof. Figure A.9 shows the full code as it is input by the user, where `Proof.` triggers the proof mode of Coq, and `Qed.` indicates that a proof is finished.

```
Lemma and_comm : A /\ B -> B /\ A.
Proof.
intros H.
destruct H as [HA HB].
split.
- apply HB.
- apply HA.
Qed.
```

Figure A.9: Full Coq proof of the commutativity of $\wedge$

Note that this is a verbose version, as Coq provides both more syntactic sugar, which reduce the quantity of text used, and tactics, which automate part of the proof. For example, Figure A.10 shows a shorter and more natural version of the same proof.

```
Lemma and_comm : A /\ B -> B /\ A.
Proof.
intro [HA HB].
split;assumption.
Qed.
```

Figure A.10: Full but shorter Coq proof of the commutativity of $\wedge$

Here, the `A /\ B` hypothesis is directly broken down into `HA` and `HB`. Just after the `B /\ A` goal is split, each generated branch calls a tactic, called `auto`. A tactic is roughly tactic, which detects that the subgoal exactly matches one of the hypotheses.

Example A.3. We now illustrate how quantifications are handled in Coq, by building a proof of $(\exists x, \forall y, P(y, x)) \to (\forall x, \exists y, P(x, y))$. The logic behind this seemingly abstruse statement is easier understood by replacing $P(x, y)$ with "y is at least as tall as x" and considering a universe of people, as it becomes *If someone is at least as tall as everyone, then, when considering any given person, you can find someone who is at least as tall as him or her.*

In Coq, we first assume an arbitrary type `A`, a binary predicate `P` over `A`, and translate our statement in Figure A.11.

```
Variable A : Type.
Variable P : A -> A -> Prop.

Lemma quant_comm : (exists x, forall y, P y x) -> (forall x, exists y, P x y).
```

Figure A.11: Writing quantifiers in Coq

We start by introducing the hypothesis. This leads us to a universally quantified statement, `forall x, exists y, P x y`. As shown in Figure A.12, we also introduce the `x` variable, meaning that we need to prove the rest of the statement with no information on the value of `x`.

```
intros H x.
```

```
H : exists x : A, forall y : A, P y x
x : A
------------------
exists y : A, P x y
```

Figure A.12: Moving a universally quantified variable to the hypotheses

Figure A.13 shows how, from the `H` hypothesis, we can extract a witness `w` that satisfies the rest of the property, which we denote as `Hw`.

```
intros H x.
destruct H as [w Hw].
```

```
w : A
Hw : forall y : A, P y w
x : A
------------------
exists y : A, P x y
```

Figure A.13: Extracing a witness from an existentially quantification

Coq now requires us to provide an explicit value for the existentially quantified `y` in the goal. We try with `w`, Coq then provides the same goal where `y` has been replaced by `w`, as shown in Figure A.14.

```
intros H x.
destruct H as [w Hw].
exists w.
```

```
w : A
Hw : forall y : A, P y w
x : A
------------------
P x w
```

Figure A.14: Instantiating an existentially quantified variable

The goal is now a special case of the `Hw` hypothesis. Figure A.15 shows how we then can finish the proof, i.e. by applying the rule with the right instance[2] of `y`.

```
intros H x.
destruct H as [w Hw].
exists w.
apply (Hw x).
```

```
No more subgoals.
```

Figure A.15: Using a universally quantified hypothesis

These examples illustrate the most simple and basic use of Coq, which also has more practical uses, such as the verification of actual algorithms and programs.

A.3 Certified programming using Coq

Coq contains a full functional programming language, called Gallina. This language, oriented towards recursion and pattern-matching, is very similar to OCaml, as illustrated by the function defined in Figure A.16.

```
Fixpoint max_in_list (l : seq nat) : nat :=
  match l with
    | [::]  => 0
    | h::tl => max h (max_in_list tl) end.
```

Figure A.16: Extracting max from a list of integers in Coq

The `max_in_list` function expects a list of integers (`seq nat`, in Coq/MathComp notations) and returns the maximal integer found in that list (or `0` if the list is empty). The `Fixpoint` construct indicates that the function is recursive, like `let rec` in OCaml.

Such functions and programs can then be caracterized using the logic language of Coq. For example, an intuitive specification of a function that extracts the maximal integer from a list would be that **1)** the extracted element does appear in the list and **2)** there is no greater example in the list. Figure A.17 shows how this specification would be written in Coq.

```
Lemma max_in_list_bad_spec :
  forall l,
       (max_in_list l \in l
    /\ forall x, x \in l -> max_in_list l <= x).
```

Figure A.17: Incorrect Coq specification of max extraction from a list

Once such a specification has been stated, it of course has to be proved, in a fashion similar to what was shown in A.2. Trying to do so for `max_in_list_bad_spec` would help the user

[2] Actually, Coq can figure it out by itself, using unification, but explicitely providing unnecessary arguments in Coq generally helps a lot understanding the high-level reasoning of a proof.

realize that he or she wrote a deficient specification, as it does not capture the case where the list is empty. Figure A.18 shows the correct specification, which would easily be proven using an induction on `l`.

```
Lemma max_in_list_spec :
  forall l, size l > 0 ->
       (max_in_list l \in l
    /\ forall x, x \in l -> max_in_list l <= x).
```

Figure A.18: Correct Coq specification of max extraction from a list

Remark A.4. Both `max_in_list` and `max_in_list_spec` are about lists of integers for simplicity, but could be generalized over the type of the list. In that case, the function would expect a specialized `max` function with the type and list, as well as a default element for the empty case. The specification would start by a universal quantification over the type, and its verification would require a proof that the provided default element is indeed minimal w.r.t. the provided `max`, just like `0` w.r.t. $\leq$.

Once the `max_in_list_spec` specification has been proved, it can be used in the certification of more complex functions built upon `max_in_list`, and so on. This way, very complex programs can be written and certified in a modular manner, as illustrated by Chapter 3, which implements a verified engine for the logic programming language Datalog. Moreover, Coq allows the extraction of such verified (or even unverified) programs into OCaml [Letouzey, 2008].

Finally, Coq only considers functions which are shown to terminate. Informally, in the framework of the Curry-Howard isomorphism upon which Coq is built, a non-terminating function is a proof of the false statement, $\bot$, which should of course not be accepted.

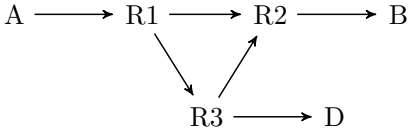
Coq can infer for itself that the program of Figure A.18 does, but some recursions are much more complex. Section 10.4.4 discusses how we had to tackle this issue for our own work.

Appendix B

Computing (very simplified) network reachability

This Appendix details the computations discussed in Section 5.2.1. The point of introducing them is to familiarize the reader with the example network, and show that even an extremely simplified, high-level view of network reachability is a surprisingly tricky problem.

The analyzed network is replicated in Figure B.1 for readability. We want to compte by hand the set of packets that will go from A to B. To do so, we will denote the *dst* and *src* attributes as a single, 6-bit vector. The first three bits represent *dst*, whereas the last three are *src*.



(a) A simple network topology ...

in	*dst*	*src*	*rewrite*	*out*
R1	10⋆	01⋆		R2
R1	1⋆⋆	⋆⋆⋆		R3
R2	10⋆	⋆⋆⋆		B
R3	⋆⋆⋆	1⋆⋆		D
R3	1⋆⋆	⋆⋆⋆	$dst[1] := 0$	R2

(b) ... and its configuration

Figure B.1: Example network

A packet flowing from A to B must first go through R1, which can be done *via* the first or second rule. In the first case, the packet header must be of the form 10⋆ 01⋆. Then, once it has reached R2, it can only go directly to B. The only relevant rule in the routing table first checks that the packet header matches 10⋆ ⋆⋆⋆, which is more general than the form of all packets going straight from R1 to R2. In a sense, we compute a conjunction, but it does not explicitely appear in the result.

We then know that all packets matching the 10⋆ 01⋆ pattern will go from A to B, using the R1 → R2 link. Another possibility is to go through R3. The first step is then to leave R1 using the R1 → R3 link, which requires using the second rule rather than the first. The set of relevant packets is those of the form 1⋆⋆ ⋆⋆⋆ but not 10⋆ 01⋆. This set is denoted as 1⋆⋆ ⋆⋆⋆ \ 10⋆ 01⋆.

We then want to use the R3 → R2 link rather than the R3 → D one. This has two consequences: 1) the leftmost bit of *src* must be 0 rather than 1 and 2) the leftmost bit of *dst* must be 1. Requirement 2) is already enforced by the R1 → R3 rule. We add 1) and obtain

the $1\star\star\ 0\star\star \backslash 10\star\ 01\star$ pattern.

The 0 on the fourth position of the right side is made obselete by the 0 just added, so we can take it off. Moreover, the central bit of *dst* is set to 0 by the rule. We add this fact and get $10\star\ 0\star\star \backslash 10\star\ \star 1\star$, which can be simplified as $10\star\ 0\star\star \backslash \star\star\star\ \star 1\star$. Since there is now only one bit defined on the right side on the pattern, we can *invert* it and put it on the left side, which gives us $10\star\ 00\star$. This pattern matches the (only) rule of R2, which leads to B.

Putting the two possibilities together, we can denote the set of packets going from A to B as

$$10\star\ 01\star \cup 10\star\ 00\star = 10\star\ 0\star\star$$

Remark B.1. The authors of [Lopes et al., 2015] denote their result as $10\star\ 01\star \cup (10\star\ \star\star\star \setminus \star\star\star\ 1\star\star)$. The two results are equivalent, as shown by the following equalities:

$$\begin{array}{lll} & 10\star\ 01\star \cup (10\star\ \star\star\star \setminus \star\star\star\ 1\star\star) & \\ = & 10\star\ 01\star \cup 10\star\ 0\star\star & \text{(integrating negation)} \\ = & 10\star\ 0\star\star & \text{(right side is more general)} \end{array}$$

We believe that the authors wanted to provide the intuition of the use of the *negation* $\setminus$, and had to make it appear in the result since they provide it directly and do not show the details of the computation.

Appendix C

Résumé français

Contexte et motivations

Au cours des dernières décennies, le monde est devenu de plus en plus numérique. Cette tendance ne s'est pas inversée en 2020 ou 2021, les services professionnels et personnels étant de plus en plus fournis et utilisés au travers d'ordinateurs, tablettes ou téléphones portables.

Cet intense *basculement numérique* implique que les pannes réseaux sont plus coûteuses et nuisibles que jamais[1], voire parfois critiques[2]. Nous insistons sur le fait que les pannes mentionnées et auxquelles nous nous intéressons ne sont pas le résultat d'attaques externes – qui par ailleurs arrivent toutes les semaines, sinon tous les jours et dans des proportions industrielles –, mais sont simplement des *bugs*.

Ces *bugs* sont avant tout dus à l'incroyable complexité de la conception de réseaux, qui elle-même vient de la nature hautement distribuée de ces derniers. De plus, la communauté réseau s'est longtemps basée sur une culture *bricolo*, dans le sens où elle ne disposait pas de fondations formelles, et donc des possibilités que l'existence et l'étude de telles fondations permet.

Durant les dix à quinze dernières années, des chercheurs et chercheuses avec un passif en théorie des langages de programmation ont commencé à s'intéresser au réseau, et à la façon dont ils pourraient appliquer leurs outils et approches théoriques à ce domaine. Combinée à l'augmentation critique des besoins en sûreté (et sécurité), cette situation a mené à l'introduction de méthodes formelles pour le réseau. Cette tendance a également été renforcée par les dernières avancées en méthodes formelles, à la fois en termes de techniques de modélisation et d'efficacité concrète (voir par exemple les *solvers* rapides comme Z3).

Parmi les outils créés, on trouve *Network Optimized Datalog* (NoD), un moteur Datalog developpé chez Microsoft conçu pour gérer des programmes qui décrivent, sous la forme de clauses de Horn, le comportement d'un réseau donné. Bien qu'étant un pas dans la bonne direction, utiliser ce moteur demande aux ingénieurs réseaux d'écrire manuellement un codage de chaque réseau analysé, ce qui est en soi un processus complexe et risqué.

De plus, NoD ne passe pas à l'échelle en utilisant des traductions *naïves* de réseaux de taille industrielle. En pratique, les auteurs se basent sur des programmes qui contiennent beaucoup de valeurs en dur, en utilisant des transformations (au niveau Datalog) manuelles, pas totalement claires et non-documentées. Cet angle mort dans un outil par ailleurs remarquable nous a poussé à travailler sur la conception et l'automatisation de transformations de programme similaires, cette fois avec une formalisation complète.

Cependant, avoir une formalisation d'opérations non-triviales n'est pas suffisant pour avoir confiance en elles. Le but de notre travail a donc été la vérification formelle de cette transformation dans l'assistant de preuve Coq, en utilisant (et étendant légèrement) une

[1] https://techcrunch.com/2020/07/17/cloudflare-dns-goes-down-taking-a-large-piece-of-the-internet-with-it/
[2] https://www.reuters.com/business/media-telecom/orange-blames-network-outage-software-failure-audit-2021-06-11/

implémentation Coq de Datalog préexistante.

Bien qu'inspiré par le cadre de la vérification réseau, notre travail n'y est pas circonscrit. Concrètement, les analyses et réécritures que nous proposons peuvent être utilisées – et pertinentes – dans d'autres contextes. De plus, nous pensons que ce travail apporte un nouvel éclairage concernant la sémantique et l'étude formelle de programmes Datalog, éclairage qui pourrait servir comme base de travaux futurs, potentiellement dans d'autres contextes.

Contribution(s)

Questions et résultats

Le point de départ de cette thèse fut l'identification et analyse d'un angle mort dans le moteur *Network Optimized Datalog* en la présence de prédicats primitifs avec plusieurs variables. Pour s'attaquer à ce problème, nous avons conçu une analyse statique pour Datalog, ainsi que deux transformations de programme sur laquelle elles se basent. L'analyse statique et les transformations ont été vérifiées dans l'assistant de preuve Coq, en se basant sur une formalisation et implémentation de Datalog dans Coq préexistante.

Notre travail a requis et mené à l'extension de certains outils, principalement l'ajout d'une sémantique de trace pour Datalog et son implémentation vérifiée dans la formalisation Coq de Datalog susmentionnée. Nous avons aussi développé des nouveaux types finis pour la bibliothèque *Mathematical Components*, sur laquelle la formalisation s'appuie.

Enfin, nous présentons une version plus fine de notre analyse statique, et montrons qu'elle ne marche pas sur tout programme Datalog. Ce résultat nous pousse à essayer de caractériser la classe précise de programmes Datalog sur lesquels elle peut être utilisée, mais notre intuition reste à vérifier formellement.

Liste de publications

Le travail sur cette thèse a mené aux contributions qui suivent :

- Une présentation au *Coq workshop 2020* sur le développement de nouveaux types finis pour la bibliothèque MathComp [Bégay et al., 2020a]
- Un papier aux $19^{èmes}$ *journées approches formelles dans l'assistance au développement de logiciels* (conférence AFADL 2020)[Bégay et al., 2020b]
- Un papier à la conférence *Certified Programs and Proofs* (CPP) 2021 [Bégay et al., 2021]

Contenu des différents chapitres

First-order logic

On rappelle les fondations de la logique du premier ordre (syntaxe et sémantique), ainsi que certains points utilisés par ailleurs dans la thèse (formes normales et systèmes d'inférence).

Datalog

Introduction formelle du langage de programmation logique Datalog, dont on définit formellement la syntaxe et deux sémantiques (dont une opérationnelle) équivalentes. On discute de plus comment traiter la négation dans Datalog, ainsi que l'utilisation de prédicats primitifs pour ajouter des calculs "non-symboliques" au langage.

Datalog in Coq

On présente d'abord succintement la bibliothèque *Mathematical Components* de Coq. On introduit ensuite la modélisation de Datalog en Coq intitulée DatalogCert, sur laquelle notre propre travail de formalistion et vérification s'appuie.

Approaches to network verification

On discute d'abord des difficultés inhérentes à la vérification réseau, puis on en présente les différents types : vérification et test du *dataplane* (d'un réseau déployé), du *control plane* (la partie automatique du déploiement d'un réseau), et enfin la synthèse de réseaux corrects par construction.

Network Optimized Datalog

Introduction de l'outil de vérification réseau NoD. On discute en particulier de l'intérêt de Datalog pour modéliser le comportement d'un réseau, et les modifications qui sont faites à un moteur Datalog préexistant pour passer à l'échelle.

Octant

On présente des limites de *Network Optimized Datalog* en termes de généricité, puis on introduit l'outil Octant qui s'y attaque. On discute ensuite du coût que le nouveau niveau de généricité a en termes d'efficacité.

New sequence and tree finTypes

On présente les nouveaux types de listes et arbres finis que nous développons dans MathComp. Les listes sont bornées soit syntaxiquement (type des listes contenant au plus x éléments), soit *sémantiquement* (type des listes sur un type fini ne contenant pas deux fois le même élément). Les arbres suivent un principe similaire (arbres bornés syntaxiquement en largeur et hauteur, ou syntaxiquement en largeur et par unicité en hauteur). Les preuves de ces types – ou plutôt de leur caractère fini – sont également présentées.

A trace semantics for Datalog

Nous introduisons une nouvelle sémantique, dite *de trace* pour Datalog. Au lieu de considérer le résultat de l'exécution d'un programme comme un ensemble de faits, nous le voyons comme un ensemble d'arbres détaillant les calculs de chacun de ces faits. Nous décrivons également la formalisation de cette sémantique dans Coq, et comment nous l'avons rendue finie pour l'intégrer au cadre de DatalogCert.

Partial program instantiation

Ce chapitre introduit notre première réécriture, qui requiert une surapproximation des substitutions calculées *in fine* par l'exécution d'un programme Datalog et s'en sert pour produire une instance partielle – et en générale beaucoup plus longue – du programme, où une partie des résultats des calculs apparaît d'office dans les règles. Le chapitre discute également de la formalisation et justification Coq de cette réécriture.

Static analysis

Nous intoduisons une analyse statique qui fournit une surapproximation du comportement de n'importe quelle variable d'un programme Datalog, et peut donc être utilisée en conjonction avec la réécriture du chapitre précédent. Fondamentalement, cette analyse représente le *chemin* parcouru par une variable lors de l'exécution comme un arbre étiqueté par des conjonctions et des disjonctions, et dont les feuilles sont des colonnes dans la base de données extensionnelles du programme analysé. Nous présentons également la formalisation et justification Coq de cette analyse statique, en particulier la difficulté liée à la preuve de terminaison.

Predicate specialization

Ce chapitre introduit notre deuxième transformation de programme, qui analyse des programmes partiellement instanciés – elle s'utilise donc en conjonction avec la première – et en spécialise les prédicats pour diminuer le nombre d'arguments manipulés. Encore une fois, la formalisation et preuve Coq sont présentées dans le chapitre.

Discussion and related works

Nous discutons certains points de notre travail sur lesquels il nous semblait intéressant de revenir. En particulier, nous commentons les effets et l'efficacité de l'optimisation dans Octant, présentons les contours d'une analyse statique plus fine qui reste à étudier plus finement, revenons sur certains choix de modélisations et erreurs faites pendant ces trois années, commentons l'effort général représenté par l'ensemble des preuves, et introduisons des travaux qui nous semblent liés à cette thèse.

Conclusion et perspectives

Le travail présenté dans ce document est, comme souvent en recherche, le résultat fortuit du *petit projet de deux semaines* proverbial. Plus concrètement, cette thèse était initialement censée s'intéresser au langage NetKat (voir la section 4.4), mais il nous a été demandé de jeter un oeil au moteur *Network Optimized Datalog* (chapitre 5) pour améliorer les performances d'Octant (chapitre 6). Le résultat est, comme souligné dans l'introduction, l'identification d'un angle mort dans le moteur susmentionné, la conception d'une analyse statique, deux transformation de programme et une sémantique de trace pour Datalog, leur certification dans une formalisation Coq/MathComp préexistente du langage et l'introduction de nouveaux types finis pour MathComp.

Nous pensons fortement qu'une des caractéristiques clefs de cette thèse est le fait qu'elle utilise DatalogCert (voir le chapitre 3), en fournissant un exemple d'utilisation non-trivial, et donc une illustration de son utilisabilité. De plus, comprendre les choix de conception d'implémentation de ce moteur, ce qui nous a mené a proposer quelques additions ou modifications, fut un exercice intellectuel très satisfaisant. Il va sans dire que nous sommes très redevables aux auteurs de DatalogCert – Ştefania-Gabriela Dumbravă, Véronique Benzaken et Evelyne Contejean –, sans qui notre travail aurait clairement manqué de fondations et justifications solides.

A un niveau plus concret, l'analyse statique et les transformataions que nous introduisons dans ce document peuvent réécrire des spécifications génériques, réutilisables, (généralement) courtes et claires en une forme spécifique à un réseau plus proche d'un programme NoD typique. Le faire à la main est bien sûr possible, mais aussi long, complexe et risqué, nos optimisations certifiées ont donc pour but de concilier performances et sûreté. Les programmes réécrits s'exécutent plusieurs ordres de grandeur plus rapidement que les originaux, mais restent parfois significativement plus lents que leurs équivalents écrits à la main dans NoD. Cette situation justifie notre travail sur une analyse statique plus fine (section 12.2), qui essaie d'échanger un peu de sûreté (il devient inadapté à certains programmes Datalog) pour beaucoup plus d'efficacité. La question clef devient alors de déterminer précisément quels programmes sont exclus – en particulier en trouvant un critère syntaxique simple –, et si les programmes restants sont pertinents. Nous donnons une esquisse de ces réponses, mais nos intuitions restent à formaliser et vérifier.

L'analyse statique et les deux réécritures que nous intoduisons sont toutes définies et certifiées indépendamment. Si une meilleure analyse statique pour Datalog que la nôtre (chapitre 10) – que ça soit la version alternative mentionnée plus haut ou une approche entièrement différente – apparaissait, elle pourrait être facilement *branchée* dans l'instance partielle de programmes du chapitre 9, notamment grâce à la très large définition de l'hypothèse de complétude de l'ensemble de substitutions fourni (voir la définition 9.4).

A l'inverse, le travail présenté dans ce document a été conçu pour *Network Optimized Data-*

log et la vérification réseau, mais pourrait être utilisé dans d'autres contextes. Cette possibilité reste à étudier.

Leçons

Pendant cette thèse, nous avons assisté à une école d'été durant laquelle Andrei Paskevich a expliqué que "La certification formelle d'un système est en général un ordre de grandeur plus complexe et longue que la conception du système en question". Ce résultat, qui pourrait au premier abord sembler abstrait, nous a finalement semblé très réél. En particulier, nous avons été étonné de constater combien des idées très simples, par exemple celles derrières l'analyse statique que nous proposons, peuvent être implémentées de façons inutilement absconses et risquées, et nécessiter un raisonnement complexe et abstrait pour les certifier.

Ces reflexions nous ramènent à la conclusion de [Benzaken et al., 2017a], qui souligne que la justification de nombreux résultats centraux et "intuitivement clairs" à propos des bases de données ont toujours été traités "de loin" plutôt qu'avec des "preuves scrupuleuses", impliquant que "des détails de bas niveaux étaient soient ignorés, soit laissés au lecteur ou la lectrice". Savoir qu'il y a une continuité et une sorte de cohérence en termes de motivations et difficultés entre l'impleméntation de DatalogCert et notre propre travail fut d'un grand réconfort.

L'autre grande leçon (ou plutôt ensemble de leçons) que nous avons apprise(s) durant ce travail est d'ordre plus qualitatif que quantitatif, et détaillée dans la section 12.3.4. Fondamentalement, nous y expliquons que

1) notre première implémentation de l'analyse statique contenait une erreur

2) et été conçue d'une façon qui rendait les preuves bien plus dures et moins claires

3) ce qui nous a mené à écrire un résultat de complètude incorrect qui a permis pendant très longtemps à une erreur de ne pas être repérée.

Nous avons suivi de nombreux cours de méthodes formelles – durant lesquels on nous a presque systématiquement raconté le crash du vol 501 d'Ariane 5 –, donc nous n'avons pas été surpris par 1). Cette situation à la "le diable est dans les détails" est, après tout, ce qui justifie les méthodes formelles.

Le leçon de 2), comme la citation de Andrei Paskevich discutée plus haut, nous la *connaissions théoriquement* mais n'en avions jamais fait l'expérience concrète. Les cours de méthodes formelles discutent bien sûr du choix de modèles et formulation appropriés, mais les systèmes et algorithmes qui y sont vérifiés sont généralement définis *a priori*. Nous n'avions donc jamais eu à travailler sur la certification d'un système mal défini ou implémenté, et ne nous attendions donc pas à ce que cet aspect soit aussi critique qu'il l'a été dans notre travail.

Enfin, le problème qui nous a le plus enseigné aura sans l'ombre d'un doute été 3). Nous avons été *élevé* – dans un sens académiquee – dans la croyance que la spécification et la vérification de propriétés d'un algorithme ou programme était un processus extrêmement puissant. Cette très forte mise en avant de la certification nous a fait croire en la *toute-*

puissance de la simple existence d'un processus de vérification, et donc nous pousser à ne pas mettre assez de soin dans ses fondations, i.e. les lemmes et théorèmes qui sont vérifiés. Tout au long du reste de notre vie en tant qu'admirateur et professionnel des méthodes formelles, nous nous souviendrons de cette expérience, afin d'atteindre et maintenir un haut niveau d'auto-critique et de perspective.

Bibliography

[rfc, 1989] (1989). Border Gateway Protocol (BGP). RFC 1105. (Cited on page 58.)

[dat, 2020] (2020). Datomic. http://www.datomic.com/. Accessed: 2020-07-16. (Cited on page 26.)

[log, 2020] (2020). Logicblox. http://www.logicblox.com/. Accessed: 2020-07-16. (Cited on page 26.)

[sem, 2020] (2020). Semmle. http://semmle.com/. Accessed: 2020-07-16. (Cited on page 26.)

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V., editors (1995). *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition. (Cited on pages 26, 33, 41, 111, and 161.)

[Abiteboul et al., 2011] Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M.-C., and Senellart, P. (2011). *Web data management*. Cambridge University Press. (Cited on page 14.)

[Al-Shaer and Al-Haj, 2010] Al-Shaer, E. and Al-Haj, S. (2010). Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44. (Cited on page 60.)

[Al-Shaer et al., 2009] Al-Shaer, E., Marrero, W., El-Atawy, A., and Elbadawi, K. (2009). Network configuration in a box: Towards end-to-end verification of network reachability and security. In *2009 17th IEEE International Conference on Network Protocols*, pages 123–132. IEEE. (Cited on page 60.)

[Anderson et al., 2014] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. (2014). Netkat: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126. (Cited on page 64.)

[Appel and Blazy, 2007] Appel, A. W. and Blazy, S. (2007). Separation logic for small-step cminor. In *International Conference on Theorem Proving in Higher Order Logics*, pages 5–21. Springer. (Cited on page 9.)

[Appel et al., 2014] Appel, A. W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., and Leroy, X. (2014). *Program Logics for Certified Compilers*. Cambridge University Press, USA. (Cited on page 9.)

[Apt et al., 1988] Apt, K. R., Blair, H. A., and Walker, A. (1988). *Towards a Theory of Declarative Knowledge*, page 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. (Cited on pages 39 and 41.)

[Aref et al., 2015] Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. L., and Washburn, G. (2015). Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382. ACM. (Cited on page 26.)

[Arntzenius and Krishnaswami, 2016] Arntzenius, M. and Krishnaswami, N. R. (2016). Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 214–227. (Cited on page 161.)

[Asperti et al., 2011] Asperti, A., Ricciotti, W., Coen, C. S., and Tassi, E. (2011). The matita interactive theorem prover. In *International Conference on Automated Deduction*, pages 64–69. Springer. (Cited on page 10.)

[Bachmair et al., 2001] Bachmair, L., Ganzinger, H., McAllester, D. A., and Lynch, C. (2001). Resolution theorem proving. In *Handbook of Automated Reasoning*. (Cited on pages 25 and 36.)

[Balbin et al., 1991] Balbin, I., Port, G., Ramamohanarao, K., and Meenakshi, K. (1991). Efficient bottom-up computation of queries on stratified databases. *The Journal of Logic Programming*, 11(3):295–344. (Cited on page 161.)

[Ball et al., 2014] Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., and Valadarsky, A. (2014). Vericon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293. (Cited on page 62.)

[Barnett et al., 2005] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2005). Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer. (Cited on page 10.)

[Barthe et al., 2017] Barthe, G., Blazy, S., Laporte, V., Pichardie, D., and Trieu, A. (2017). Verified translation validation of static analyses. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 405–419. IEEE. (Cited on page 162.)

[Barthe et al., 2013] Barthe, G., Pichardie, D., and Rezk, T. (2013). A certified lightweight non-interference Java bytecode verifier. *Mathematical Structures in Computer Science (MSCS)*, 23(5):1032–1081. (Cited on page 162.)

[Bauer et al., 2017] Bauer, A., Gross, J., Lumsdaine, P. L., Shulman, M., Sozeau, M., and Spitters, B. (2017). The hott library: a formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 164–172. (Cited on page 9.)

[Bauer et al., 2011] Bauer, L., Garriss, S., and Reiter, M. K. (2011). Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):1–28. (Cited on page 62.)

[Beckett et al., 2018] Beckett, R. et al. (2018). Network control plane synthesis and verification. (Cited on pages 61 and 62.)

[Beckett et al., 2016] Beckett, R., Greenberg, M., and Walker, D. (2016). Temporal netkat. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 386–401. (Cited on page 64.)

[Beckett et al., 2017] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. (2017). A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168. (Cited on page 62.)

[Beeson et al., 2018] Beeson, M., Boutry, P., Braun, G., Gries, C., and Narboux, J. (2018). GeoCoq. (Cited on page 9.)

[Bégay et al., 2020a] Bégay, P.-L., Crégut, P., and Monin, J.-F. (2020a). Developing sequence and tree fintypes in MathComp. Coq Workshop 2020. (Cited on pages 8 and 178.)

[Bégay et al., 2020b] Bégay, P.-L., Crégut, P., and Monin, J.-F. (2020b). Octant, la vérification réseau simplifiée. In *19èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL 2020*, Vannes, France. (Cited on pages 8 and 178.)

[Bégay et al., 2021] Bégay, P.-L., Crégut, P., and Monin, J.-F. (2021). Developing and certifying Datalog optimizations in Coq/Mathcomp. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 163–177, New York, NY, USA. Association for Computing Machinery. (Cited on pages 8 and 178.)

[Benzaken et al., 2017a] Benzaken, V., Contejean, É., and Dumbrava, S. (2017a). Certifying Standard and Stratified Datalog Inference Engines in SSReflect. In *International Conference on Interective Theorem Proving*, Brasilia, Brazil. (Cited on pages 26, 44, 158, 165, and 184.)

[Benzaken et al., 2017b] Benzaken, V., Contejean, É., and Dumbrava, S. (2017b). DatalogCert. https://framagit.org/formaldata/datalogcert/. (Cited on pages 13, 37, 41, 44, 45, 47, 78, 153, and 159.)

[Besson et al., 2006] Besson, F., Jensen, T., and Pichardie, D. (2006). Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273–291. (Cited on page 162.)

[Biere et al., 2009] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. (Cited on page 65.)

[Björklund, 2010] Björklund, M. (2010). YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020. (Cited on page 63.)

[Bjørner and Jayaraman, 2015] Bjørner, N. and Jayaraman, K. (2015). Checking cloud contracts in microsoft azure. In *International Conference on Distributed Computing and Internet Technology*, pages 21–32. Springer. (Cited on page 61.)

[Bjørner et al., 2015] Bjørner, N., Phan, A.-D., and Fleckenstein, L. (2015). νz-an optimizing smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer. (Cited on page 10.)

[Bjørner and Varghese, 2015] Bjørner, N. and Varghese, G. (2015). Network verification: when hoare meets cerf. ACM SIGCOMM Tutorial. (Cited on page 61.)

[Bodin et al., 2018] Bodin, M., Diaz, T., and Tanter, E. (2018). A trustworthy mechanized formalization of r. *SIGPLAN Not.*, 53(8):13–24. (Cited on page 9.)

[Bonifati et al., 2018] Bonifati, A., Dumbrava, S., and Arias, E. J. G. (2018). Certified graph view maintenance with regular datalog. *Theory Pract. Log. Program.*, 18(3-4):372–389. (Cited on pages 44 and 162.)

[Bosshart et al., 2014] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95. (Cited on page 64.)

[Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691. (Cited on pages 60 and 70.)

[Caballero et al., 2008] Caballero, R., García-Ruiz, Y., and Sáenz-Pérez, F. (2008). A new proposal for debugging datalog programs. *Electronic Notes in Theoretical Computer Science*, 216:79–92. (Cited on page 161.)

[Caballero et al., 2007] Caballero, R., Hermanns, C., and Kuchen, H. (2007). Algorithmic debugging of java programs. *Electronic Notes in Theoretical Computer Science*, 177:75–89. Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP 2006). (Cited on page 161.)

[Cachera et al., 2004] Cachera, D., Jensen, T., Pichardie, D., and Rusu, V. (2004). Extracting a data flow analyser in constructive logic. In *European Symposium on Programming*, pages 385–400. Springer. (Cited on page 162.)

[Cadar et al., 2008] Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224. (Cited on pages 10 and 61.)

[Calì et al., 2009] Calì, A., Gottlob, G., and Lukasiewicz, T. (2009). Datalog±: a unified approach to ontologies and integrity constraints. In *Proceedings of the 12th International Conference on Database Theory*, pages 14–30. ACM. (Cited on page 26.)

[Campagna et al., 2011] Campagna, D., Sarna-Starosta, B., and Schrijvers, T. (2011). Approximating constraint propagation in Datalog. *CoRR*, abs/1112.3787. (Cited on page 160.)

[Canini et al., 2012] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. (2012). A NICE way to test openflow applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140. (Cited on page 62.)

[Casado et al., 2007] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. (2007). Ethane: Taking control of the enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12. (Cited on page 63.)

[Castéran and Bertot, 2004] Castéran, P. and Bertot, Y. (2004). *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Texts in Theoretical Computer Science. Springer Verlag. Traduction en chinois parue en 2010. Tsinghua University Press. ISBN 9787302208136. (Cited on page 9.)

[Ceri et al., 1989] Ceri, S., Gottlob, G., and Tanca, L. (1989). What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166. (Cited on page 35.)

[Chang et al., 2003] Chang, D.-F., Govindan, R., and Heidemann, J. (2003). The temporal and topological characteristics of BGP path changes. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, pages 190–199. IEEE. (Cited on page 62.)

[Chaudhuri, 1993] Chaudhuri, S. (1993). Finding nonrecursive envelopes for Datalog predicate. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '93, page 135–146, New York, NY, USA. Association for Computing Machinery. (Cited on page 160.)

[Chaudhuri and Kolaitis, 1994] Chaudhuri, S. and Kolaitis, P. G. (1994). Can Datalog be approximated? In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '94, page 86–96, New York, NY, USA. Association for Computing Machinery. (Cited on page 160.)

[Chin et al., 2015] Chin, B., von Dincklage, D., Ercegovac, V., Hawkins, P., Miller, M. S., Och, F., Olston, C., and Pereira, F. (2015). Yedalog: Exploring knowledge at scale. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. (Cited on page 26.)

[Chlipala, 2010] Chlipala, A. (2010). A verified compiler for an impure functional language. *ACM Sigplan Notices*, 45(1):93–106. (Cited on page 9.)

[Cimatti et al., 2000] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. (2000). Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425. (Cited on page 60.)

[Clark, 1988] Clark, D. (1988). The design philosophy of the DARPA internet protocols. In *Symposium proceedings on Communications architectures and protocols*, pages 106–114. (Cited on page 57.)

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263. (Cited on page 60.)

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387. (Cited on page 26.)

[Codd, 2002] Codd, E. F. (2002). A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer. (Cited on page 14.)

[Constable et al., 1986] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., USA. (Cited on page 10.)

[Cousot, 2002] Cousot, P. (2002). Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1-2):47–103. (Cited on page 90.)

[Cousot, 2005] Cousot, P. (2005). Abstract interpretation. MIT course 16.399, http://web.mit.edu/16.399/www/. (Cited on page 90.)

[Cruz-Filipe et al., 2004] Cruz-Filipe, L., Geuvers, H., and Wiedijk, F. (2004). C-corn, the constructive Coq repository at Nijmegen. In *International Conference on Mathematical Knowledge Management*, pages 88–103. Springer. (Cited on page 9.)

[de Moor et al., 2008] de Moor, O., Sereni, D., Avgustinov, P., and Verbaere, M. (2008). Type inference for Datalog and its application to query optimisation. In *Proceedings of the*

Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '08, page 291–300, New York, NY, USA. Association for Computing Machinery. (Cited on page 160.)

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340. (Cited on pages 10, 60, and 65.)

[de Rauglaudre, 2017] de Rauglaudre, D. (2017). Formal Proof of Banach-Tarski Paradox. *Journal of Formalized Reasoning*, 10(1):37–49. (Cited on page 9.)

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113. (Cited on page 14.)

[DeTreville, 2002] DeTreville, J. (2002). Binder, a logic-based security language. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE. (Cited on page 26.)

[Dobrescu and Argyraki, 2014] Dobrescu, M. and Argyraki, K. (2014). Software dataplane verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 101–114. (Cited on page 61.)

[Doenges et al., 2021] Doenges, R., Arashloo, M. T., Bautista, S., Chang, A., Ni, N., Parkinson, S., Peterson, R., Solko-Breslin, A., Xu, A., and Foster, N. (2021). Petr4: formal foundations for p4 data planes. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32. (Cited on pages 57, 58, and 64.)

[Dougherty et al., 2006] Dougherty, D. J., Fisler, K., and Krishnamurthi, S. (2006). Specifying and reasoning about dynamic access-control policies. In *International Joint Conference on Automated Reasoning*, pages 632–646. Springer. (Cited on page 29.)

[Dumbrava, 2016] Dumbrava, S.-G. (2016). *Formalisation en Coq de Bases de Données Relationnelles et Déductives -et Mécanisation de Datalog*. PhD thesis, Université Paris-Sud. (Cited on pages 10, 13, 28, 36, 37, 40, 41, 44, 47, 52, 78, 153, 155, and 164.)

[Duschka, 1998] Duschka, O. M. (1998). *Query Planning and Optimization in Information Integration*. PhD thesis, Stanford, CA, USA. AAI9837087. (Cited on page 160.)

[Enns et al., 2011] Enns, R., Björklund, M., Bierman, A., and Schönwälder, J. (2011). Network Configuration Protocol (NETCONF). RFC 6241. (Cited on page 63.)

[Fayaz et al., 2016] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. (2016). Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 217–232. (Cited on page 59.)

[Feamster and Balakrishnan, 2005] Feamster, N. and Balakrishnan, H. (2005). Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56. (Cited on page 62.)

[Fogel et al., 2015] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. (2015). A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483. (Cited on page 62.)

[Foster et al., 2011] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. (2011). Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291. (Cited on page 63.)

[Foster et al., 2016] Foster, N., Kozen, D., Mamouras, K., Reitblatt, M., and Silva, A. (2016). Probabilistic netkat. In *European Symposium on Programming*, pages 282–309. Springer. (Cited on page 64.)

[Foster et al., 2015] Foster, N., Kozen, D., Milano, M., Silva, A., and Thompson, L. (2015). A coalgebraic decision procedure for netkat. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–355. (Cited on page 64.)

[Gao and Rexford, 2001] Gao, L. and Rexford, J. (2001). Stable internet routing without global coordination. *IEEE/ACM Transactions on networking*, 9(6):681–692. (Cited on page 62.)

[Garillot et al., 2009] Garillot, F., Gonthier, G., Mahboubi, A., and Rideau, L. (2009). Packaging Mathematical Structures. In Nipkow, T. and Urban, C., editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany. Springer. (Cited on pages 44 and 46.)

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080. (Cited on page 41.)

[Gember-Jacobson et al., 2016] Gember-Jacobson, A., Viswanathan, R., Akella, A., and Mahajan, R. (2016). Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 300–313. (Cited on page 62.)

[Gonthier, 2007] Gonthier, G. (2007). The four colour theorem: Engineering of a formal proof. In *Asian Symposium on Computer Mathematics*, pages 333–333. Springer. (Cited on page 9.)

[Gonthier et al., 2016] Gonthier, G., Mahboubi, A., and Tassi, E. (2016). A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France. (Cited on page 9.)

[Gottlob et al., 2004] Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., and Flesca, S. (2004). The lixto data extraction project: back and forth between theory and practice. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12. ACM. (Cited on page 26.)

[Green and Raphael, 1967] Green, C. C. and Raphael, B. (1967). Research on intelligent question-answering system. Technical report, STANFORD RESEARCH INST MENLO PARK CA. (Cited on page 14.)

[Greenman, 2017] Greenman, B. (2017). Datalog for static analysis. (Cited on pages 26 and 161.)

[Griffin and Wilfong, 1999] Griffin, T. G. and Wilfong, G. (1999). An analysis of BGP convergence properties. *ACM SIGCOMM Computer Communication Review*, 29(4):277–288. (Cited on page 62.)

[Grumbach and Wang, 2010] Grumbach, S. and Wang, F. (2010). Netlog, a rule-based language for distributed programming. In *International Symposium on Practical Aspects of Declarative Languages*, pages 88–103. Springer. (Cited on page 26.)

[Guha et al., 2013] Guha, A., Reitblatt, M., and Foster, N. (2013). Machine-verified network controllers. *Acm Sigplan Notices*, 48(6):483–494. (Cited on page 63.)

[Halevy et al., 2001] Halevy, A. Y., Mumick, I. S., Sagiv, Y., and Shmueli, O. (2001). Static analysis in Datalog extensions. *J. ACM*, 48(5):971–1012. (Cited on pages 91 and 160.)

[Harrison, 2013] Harrison, J. (2013). The HOL Light theory of euclidean space. *Journal of Automated Reasoning*, 50:173–190. (Cited on page 9.)

[Hellerstein, 2010] Hellerstein, J. M. (2010). The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record*, 39(1):5–19. (Cited on page 26.)

[Herbrand, 1930] Herbrand, J. (1930). *Recherches sur la théorie de la démonstration*. Doctorat d'état. (Cited on page 14.)

[Heyd, 1997] Heyd, B. (1997). *Application de la théorie des types et du démonstrateur Coq à la vérification de programmes parallèles*. Theses, Université Henri Poincaré - Nancy 1. Texte intégral accessible uniquement aux membres de l'Université de Lorraine. (Cited on page 62.)

[Hoare, 1978] Hoare, C. A. R. (1978). Some properties of predicate transformers. *Journal of the ACM (JACM)*, 25(3):461–480. (Cited on page 90.)

[Hoder and Bjørner, 2012] Hoder, K. and Bjørner, N. (2012). Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer. (Cited on page 10.)

[Hoder et al., 2011] Hoder, K., Bjørner, N., and de Moura, L. (2011). μz–an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462. Springer. (Cited on pages 10, 65, 70, and 76.)

[Hu et al., 2014] Hu, F., Hao, Q., and Bao, K. (2014). A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys & Tutorials*, 16(4):2181–2206. (Cited on page 57.)

[Huang et al., 2011] Huang, S. S., Green, T. J., and Loo, B. T. (2011). Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216. ACM. (Cited on page 26.)

[Jacobs and Langen, 1992] Jacobs, D. and Langen, A. (1992). Static analysis of logic programs for independent and parallelism. *The Journal of Logic Programming*, 13(2-3):291–314. (Cited on page 160.)

[Jard et al., 1987] Jard, C., Monin, J.-F., and Groz, R. (1987). Development of Veda, a prototyping tool for distributed algorithms. Research Report RT-0087, INRIA. (Cited on page 62.)

[Jayaraman et al., 2014] Jayaraman, K., Bjørner, N., Outhred, G., and Kaufman, C. (2014). Automated analysis and debugging of network connectivity policies. Technical Report MSR-TR-2014-102, Microsoft. (Cited on page 61.)

[Jeannet and Serwe, 2004] Jeannet, B. and Serwe, W. (2004). Abstracting call-stacks for interprocedural verification of imperative programs. In *International Conference on Algebraic Methodology and Software Technology*, pages 258–273. Springer. (Cited on pages 125 and 162.)

[Jhala and Majumdar, 2009] Jhala, R. and Majumdar, R. (2009). Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54. (Cited on page 65.)

[Kazemian et al., 2013] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. (2013). Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111. (Cited on pages 61 and 69.)

[Kazemian et al., 2012] Kazemian, P., Varghese, G., and McKeown, N. (2012). Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126. (Cited on pages 60 and 69.)

[Kessens et al., 1999] Kessens, D., Bates, T. J., Alaettinoglu, C., Meyer, D., Villamizar, C., Terpstra, M., Karrenberg, D., and Gerich, E. P. (1999). Routing Policy Specification Language (RPSL). RFC 2622. (Cited on page 63.)

[Khurshid et al., 2013] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. (2013). Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27. (Cited on pages 61, 65, and 69.)

[Kozen and Smith, 1996] Kozen, D. and Smith, F. (1996). Kleene algebra with tests: Completeness and decidability. In *International Workshop on Computer Science Logic*, pages 244–259. Springer. (Cited on page 64.)

[Kriener et al., 2013] Kriener, J., King, A., and Blazy, S. (2013). Proofs you can believe in. proving equivalences between prolog semantics in Coq. pages 37–48. (Cited on page 162.)

[Kuhns, 1967] Kuhns, J. (1967). Answering questions by computer: a logical study. Technical report, RAND CORP SANTA MONICA CA SANTA MONICA United States. (Cited on page 14.)

[Kumar et al., 2014] Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). Cakeml: a verified implementation of ml. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM. (Cited on pages 9 and 162.)

[Kwiatkowska et al., 2011] Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of Probabilistic Real-time Systems. In Gopalakrishnan, G. and Qadeer, S., editors, *23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591, Snowbird, United States. Springer. (Cited on page 61.)

[Lakshminarayanan et al., 2005] Lakshminarayanan, K., Rangarajan, A., and Venkatachary, S. (2005). Algorithms for advanced packet classification with ternary cams. *ACM SIGCOMM Computer Communication Review*, 35(4):193–204. (Cited on page 66.)

[Larchey-Wendling and Monin, 2018] Larchey-Wendling, D. and Monin, J.-F. (2018). Simulating Induction-Recursion for Partial Algorithms. In *24th International Conference on Types for Proofs and Programs,TYPES 2018*, Braga, Portugal. (Cited on page 129.)

[Le et al., 2008] Le, F., Xie, G. G., Pei, D., Wang, J., and Zhang, H. (2008). Shedding light on the glue logic of the internet routing architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 39–50. (Cited on page 62.)

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115. (Cited on pages 9 and 162.)

[Leroy, 2019] Leroy, X. (2019). *Le logiciel, entre l'esprit et la matière*. Leçon inaugurale - Collège de France. Fayard. (Cited on page 14.)

[Letan and Régis-Gianas, 2020] Letan, T. and Régis-Gianas, Y. (2020). Freespec: Specifying, verifying, and executing impure computations in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 32–46, New York, NY, USA. Association for Computing Machinery. (Cited on page 9.)

[Letouzey, 2008] Letouzey, P. (2008). Extraction in Coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer. (Cited on pages 162 and 172.)

[Levien and Maron, 1965] Levien, R. and Maron, M. (1965). Relational data file: a tool for mechanized inference execution and data retrieval. Technical report, RAND CORP SANTA MONICA CA. (Cited on page 14.)

[Li et al., 2000] Li, T., Farinacci, D., Hanks, S. P., Meyer, D., and Traina, P. S. (2000). Generic Routing Encapsulation (GRE). RFC 2784. (Cited on page 58.)

[Liu et al., 2017] Liu, H. H., Zhu, Y., Padhye, J., Cao, J., Tallapragada, S., Lopes, N. P., Rybalchenko, A., Lu, G., and Yuan, L. (2017). Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613. (Cited on page 62.)

[Liu, 1999] Liu, M. (1999). Deductive database languages: problems and solutions. *ACM Computing Surveys (CSUR)*, 31(1):27–62. (Cited on page 26.)

[Lloyd, 1987a] Lloyd, J. W. (1987a). *Foundations of Logic Programming*. Springer. (Cited on page 161.)

[Lloyd, 1987b] Lloyd, J. W. (1987b). *Foundations of Logic Programming, 2nd Edition*. Springer. (Cited on page 13.)

[Loo et al., 2005] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. (2005). Implementing declarative overlays. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 75–90. ACM. (Cited on page 26.)

[Lopes,] Lopes, N. P. Network verification website (benchmarks and code). http://web.ist.utl.pt/nuno.lopes/netverif/. (Cited on page 146.)

[Lopes et al., 2015] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., and Varghese, G. (2015). Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA. USENIX Association. (Cited on pages 10, 60, 65, 66, 67, 68, 69, 70, 74, and 174.)

[Lopes et al., 2013] Lopes, N. P., Bjorner, N., Godefroid, P., and Varghese, G. (2013). Network verification in the light of program verification. Technical report, Microsoft. (Cited on pages 65 and 70.)

[Lu and Cleary, 1999] Lu, L. and Cleary, J. G. (1999). An operational semantics of starlog. In *International Conference on Principles and Practice of Declarative Programming*, pages 294–310. Springer. (Cited on page 26.)

[Madsen et al., 2016] Madsen, M., Yee, M.-H., and Lhoták, O. (2016). From Datalog to Flix: A declarative language for fixed points on lattices. *SIGPLAN Not.*, 51(6):194–208. (Cited on page 161.)

[Mai et al., 2011] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. (2011). Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301. (Cited on page 60.)

[Makarov and Spitters, 2013] Makarov, E. and Spitters, B. (2013). The Picard algorithm for ordinary differential equations in Coq. In *International Conference on Interactive Theorem Proving*, pages 463–468. Springer. (Cited on page 9.)

[Marriott et al., 1994] Marriott, K., Søndergaard, H., and Jones, N. D. (1994). Denotational abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.*, 16(3):607–648. (Cited on page 160.)

[McKeown et al., 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74. (Cited on page 58.)

[Mesnard and Neumerkel, 2001] Mesnard, F. and Neumerkel, U. (2001). Applying static analysis techniques for inferring termination conditions of logic programs. In Cousot, P., editor, *Static Analysis*, pages 93–110, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 160.)

[Might et al., 2010] Might, M., Smaragdakis, Y., and Van Horn, D. (2010). Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315. (Cited on page 161.)

[Miller, 2006] Miller, D. (2006). Collection analysis for horn clause programs. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 179–188. (Cited on page 160.)

[Miller, 2008] Miller, D. (2008). A proof-theoretic approach to the static analysis of logic programs. *Reasoning in Simple Type Theory: Festschrift in honour of Peter B. Andrews on his 70th birthday. Studies in Logic*, 17. (Cited on page 160.)

[Minker, 1988] Minker, J. (1988). Perspectives in deductive databases. *The Journal of Logic Programming*, 5(1):33–60. (Cited on page 14.)

[Monin, 1989] Monin, J.-F. (1989). *Programmation en logique et compilation de protocoles : le simulateur Véda*. Thèse d'université, Université de Rennes I. (Cited on page 62.)

[Monniaux, 2016] Monniaux, D. (2016). A survey of satisfiability modulo theory. In *International Workshop on Computer Algebra in Scientific Computing*, pages 401–425. Springer. (Cited on page 10.)

[Monsanto et al., 2012] Monsanto, C., Foster, N., Harrison, R., and Walker, D. (2012). A compiler and run-time system for network programming languages. *Acm sigplan notices*, 47(1):217–230. (Cited on page 63.)

[Monsanto et al., 2013] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. (2013). Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13. (Cited on page 63.)

[Nadathur and Miller, 1988] Nadathur, G. and Miller, D. (1988). An overview of lambda-prolog. (Cited on page 160.)

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer. (Cited on page 9.)

[Norell, 2008] Norell, U. (2008). Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer. (Cited on page 10.)

[O Connor, 2005] O Connor, R. (2005). Essential incompleteness of arithmetic verified by Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 245–260. Springer. (Cited on page 9.)

[Owre et al., 1992] Owre, S., Rushby, J. M., and Shankar, N. (1992). Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer. (Cited on page 10.)

[Perera et al., 2012] Perera, R., Acar, U. A., Cheney, J., and Levy, P. B. (2012). Functional programs that explain their work. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 365–376. (Cited on page 91.)

[Plotkin et al., 2016] Plotkin, G. D., Bjørner, N., Lopes, N. P., Rybalchenko, A., and Varghese, G. (2016). Scaling network verification using symmetry and surgery. *ACM SIGPLAN Notices*, 51(1):69–83. (Cited on page 61.)

[Przymusinski, 1990] Przymusinski, T. (1990). Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, 13:445–463. (Cited on page 42.)

[Przymusinski, 1988] Przymusinski, T. C. (1988). On the declarative semantics of deductive databases and logic programs. In *Foundations of deductive databases and logic programming*, pages 193–216. Elsevier. (Cited on page 41.)

[Queille and Sifakis, 1982] Queille, J.-P. and Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer. (Cited on page 62.)

[Ramakrishnan and Ullman, 1995] Ramakrishnan, R. and Ullman, J. D. (1995). A survey of deductive database systems. *The journal of logic programming*, 23(2):125–149. (Cited on page 26.)

[Rekhter et al., 2006] Rekhter, Y., Li, T., and Hares, S. (2006). A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard). (Cited on page 62.)

[Reutter et al., 2015] Reutter, J. L., Romero, M., and Vardi, M. Y. (2015). Regular Queries on Graph Databases. In Arenas, M. and Ugarte, M., editors, *18th International Conference on Database Theory (ICDT 2015)*, volume 31 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 177–194, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cited on page 162.)

[Robinson, 1974] Robinson, J. A. (1974). Automatic deduction with hyper-resolution. (Cited on pages 14 and 25.)

[Russell and Norvig, 2009] Russell, S. J. and Norvig, P. (2009). *Artificial Intelligence: a modern approach*. Pearson, 3 edition. (Cited on page 23.)

[Sakaguchi, 2020] Sakaguchi, K. (2020). Validating mathematical structures. *Lecture Notes in Computer Science*, page 138–157. (Cited on page 44.)

[Scholz et al., 2016] Scholz, B., Jordan, H., Subotić, P., and Westmann, T. (2016). On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA. ACM. (Cited on page 161.)

[Sefraoui et al., 2012] Sefraoui, O., Aissaoui, M., and Eleuldj, M. (2012). Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42. (Cited on pages 10 and 72.)

[Seo et al., 2013] Seo, J., Park, J., Shin, J., and Lam, M. S. (2013). Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917. (Cited on page 26.)

[Shenker et al.,] Shenker, S. et al. The future of networking, and the past of protocols. (Cited on page 58.)

[Shivers, 1991] Shivers, O. (1991). *Control-flow analysis of higher-order languages*. PhD thesis. (Cited on pages 125 and 161.)

[Smolka et al., 2015] Smolka, S., Eliopoulos, S., Foster, N., and Guha, A. (2015). A fast compiler for netkat. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 328–341. (Cited on page 64.)

[Sozeau, 2010] Sozeau, M. (2010). Equations: A dependent pattern-matching compiler. In *International Conference on Interactive Theorem Proving*, pages 419–434. Springer. (Cited on page 79.)

[Swamy et al., 2013] Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., and Livshits, B. (2013). Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398. (Cited on page 10.)

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309. (Cited on page 30.)

[Tristan and Leroy, 2008] Tristan, J.-B. and Leroy, X. (2008). Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–27. (Cited on pages 155 and 158.)

[Ullman, 1990] Ullman, J. D. (1990). Principles of database and knowledge-base systems, volume ii: The new technologies. (Cited on page 40.)

[Van Emden and Kowalski, 1976] Van Emden, M. H. and Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742. (Cited on pages 14, 26, 30, and 35.)

[Van Gelder et al., 1991] Van Gelder, A., Ross, K. A., and Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3):619–649. (Cited on page 42.)

[Viswanathan et al., 2001] Viswanathan, A., Rosen, E. C., and Callon, R. (2001). Multiprotocol Label Switching Architecture. RFC 3031. (Cited on pages 58 and 69.)

[Weber et al., 2019] Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., and Reger, G. (2019). The smt competition 2015–2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):221–259. (Cited on page 10.)

[Weitz et al., 2016] Weitz, K., Woos, D., Torlak, E., Ernst, M. D., Krishnamurthy, A., and Tatlock, Z. (2016). Formal semantics and automated verification for the border gateway protocol. *NetPL, March*. (Cited on page 62.)

[Whaley et al., 2005] Whaley, J., Avots, D., Carbin, M., and Lam, M. S. (2005). Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer. (Cited on pages 26 and 161.)

[Winterhalter, 2020] Winterhalter, T. (2020). *Formalisation and Meta-Theory of Type Theory*. PhD thesis. Thèse de doctorat dirigée par Tabareau, Nicolas et Sozeau, Matthieu Informatique Nantes 2020. (Cited on page 166.)

[Wintersteiger et al., 2013] Wintersteiger, C. M., Hamadi, Y., and De Moura, L. (2013). Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23. (Cited on page 10.)

[Xie et al., 2005] Xie, G. G., Zhan, J., Maltz, D. A., Zhang, H., Greenberg, A., Hjalmtysson, G., and Rexford, J. (2005). On static reachability analysis of ip networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 2170–2183. IEEE. (Cited on page 60.)

[Yousefi et al., 2020] Yousefi, F., Abhashkumar, A., Subramanian, K., Hans, K., Ghorbani, S., and Akella, A. (2020). Liveness verification of stateful network functions. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 257–272. (Cited on page 61.)

[Zeng et al., 2012] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. (2012). Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. (Cited on page 61.)

[Zhang et al., 2021] Zhang, H., Zhang, C., Azevedo de Amorim, A., Agarwal, Y., Fredrikson, M., and Jia, L. (2021). Netter: Probabilistic, stateful network models. In Henglein, F., Shoham, S., and Vizel, Y., editors, *Verification, Model Checking, and Abstract Interpretation*, pages 486–508, Cham. Springer International Publishing. (Cited on page 61.)

[Zhang et al.,] Zhang, S., Mahmoud, A., Malik, S., and Narain, S. Verification and synthesis of firewalls using sat and qbf. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE. (Cited on page 60.)

[Zhang and Malik, 2013] Zhang, S. and Malik, S. (2013). Sat based verification of network data planes. In *Automated Technology for Verification and Analysis*, pages 496–505. Springer. (Cited on page 60.)

[Zook et al., 2009] Zook, D., Pasalic, E., and Sarna-Starosta, B. (2009). Typed datalog. In *International Symposium on Practical Aspects of Declarative Languages*, pages 168–182. Springer. (Cited on page 160.)

[Zsidó, 2013] Zsidó, J. (2013). Theorem of three circles in Coq. *Journal of Automated Reasoning*. 25 pages, 5 figures. (Cited on page 9.)