Formalisation en Coq de Bases de Données Relationnelles et Déductives -et Mécanisation de Datalog

Ştefania-Gabriela Dumbravă

NNT : 2016SACLS525, Code INE : 0jj0dm01px

Thèse de doctorat
de
l'Université Paris-Saclay préparée à
l'Université Paris-Sud

École doctorale 580
Sciences et Technologies de l'Information et de la Communication

Spécialité Informatique

par

Mme Ștefania-Gabriela Dumbravă

A Coq Formalization of Relational and Deductive Databases - and Mechanizations of Datalog

Thèse présentée et soutenue à Orsay, le vendredi 2 décembre 2016

Composition du Jury :

Mme Véronique Benzaken	Professeur à l'Université Paris-Sud	Directrice
Mme Sandrine Blazy	Professeur à l'Université de Rennes 1	Rapportrice
Mme Sarah Cohen-Boulakia	Maître de Conferences (HDR) à l'Université Paris-Sud	Examinatrice
Mme Évelyne Contejean	Chargée de Recherche (HDR) CNRS	Directrice
M. Mohand-Saïd Hacid	Professeur à l'Université Claude Bernard Lyon 1	Rapporteur
M. Michaël Rusinowitch	Directeur de Recherche à l'INRIA Lorraine	President du Jury

université PARIS-SACLAY | ÉCOLE DOCTORALE Sciences et technologies de l'information et de la communication (STIC)

Titre :

Formalisation en Coq de Bases de Données Relationnelles et Déductives - et Mécanisations de Datalog

Mots-clefs :

méthodes formelles, assistants de preuve, logique, bases de données

Dans plusieurs domaines de l'informatique, des techniques basées sur les tests, la modélisation formelle ou la vérification ont été employées avec succès pour aider les programmeurs à créer des systèmes fiables. Par exemple, dans le développement des processeurs, les prouveurs automatiques de théorèmes révèlent des bugs profonds dans le design avant qu'ils ne deviennent des erreurs coûteuses en silicone ; les développeurs en avionique utilisent l'analyse des programme pour vérifier des propriétés de sûreté critiques dans les logiciels embarqués des avions ; les vendeurs des systèmes d'exploitation utilisent le "model checking" pour éliminer des bugs dans des pilotes de périphérique. Pourtant, jusqu'à récemment, les systèmes de gestion des données n'ont pas utilisé les moyens d'analyse fournis par les méthodes formelles.

De plus, les systèmes de gestion des données actuels manipulent des volumes de données de plus en plus massifs. Ces données sont précieuses et leur disponibilité, intégrité et fiabilité sont essentielles pour les entreprises, les scientifiques et les citoyens. En conséquence, il est important d'assurer des garanties fortes de ces propriétés en établissant que ces systèmes eux-mêmes sont sûrs et fiables. Pour atteindre cet objectif, des méthodes formelles et des outils matures doivent être utilisées. Une approche très prometteuse consiste en l'utilisation des assistants de preuve comme Coq.

La vérification et la certification des programmes ont été intensivement étudiés dans ces dernières années et ont produit des résultats impressionnants et des logiciels fiables. De manière surprenante, malgré le fait que le montant des données stockées et manipulées par les moteurs de données a augmenté, peu d'attention a été accordée pour assurer la fiabilité de ces systèmes. Parmi eux, les systèmes relationnels de gestion sont les plus répandus. Par ailleurs, les systèmes déductifs fournissent un cadre unificateur pour une multitude de langages de requêtes et resurgissent dans le contexte du Web Sémantique. Par conséquent, ces observations justifient notre choix de formaliser le modèle relationnel, ainsi que le modèle déductif - restreint au cas du Datalog.

Cette thèse présente une formalisation en Coq des langages et des algorithmes fondamentaux portant sur les bases de données. Cela fourni des spécifications formelles issues des deux approches différentes pour la définition des modèles de données : une basée sur l'algèbre et l'autre basée sur la logique.

A ce titre, une première contribution de cette thèse est le développement d'une bibliothèque Coq pour le modèle relationnel. Celui-ci contient des modélisations de l'algèbre relationnelle et des requêtes conjonctives. Il contient aussi une mécanisation des contraintes d'intégrité et de leurs procédures d'inférence. Nous modélisons deux types des dépendances, qui sont parmi

les plus courantes : les dépendances fonctionnelles et les dépendances multivaluées, ainsi que leurs axiomatisations correspondants. Nous prouvons formellement la correction de leurs algorithmes d'inférence et, pour le cas de dépendances fonctionnelles, aussi la complétude. Ces types de dépendances sont des instances de dépendances générales : les dépendances génératrices d'égalité et, respectivement, les dépendances génératrices de tuples. Nous modélisons les dépendances générales et leur procédure d'inférence, c'est-à-dire, "le chase", pour lequel nous établissons la correction. Enfin, on prouve formellement les théorèmes principaux des bases de données, c'est-à-dire, les équivalences algébriques, le théorème de l'homomorphisme et la minimisation des requêtes conjonctives.

Une deuxième contribution consiste dans le développement d'une bibliothèque Coq/ssreflect pour la programmation logique, restreinte au cas du Datalog. Dans le cadre de ce travail, nous donnons la première mécanisation d'un moteur Datalog standard, ainsi que son extension avec la négation. La bibliothèque comprend une formalisation de leur sémantique en théorie des modèles ainsi que de leur sémantique par point fixe, implémentée par une procédure d'évaluation stratifiée. La bibliothèque est complétée par les preuves de correction, de terminaison et de complétude correspondantes. Dans ce contexte, nous construisons aussi un cadre préliminaire pour raisonner sur les programmes stratifiés. Cette plateforme ouvre la voie à la certification d'applications centrées données.

Title :

A Coq Formalization of Relational and Deductive Databases - and Mechanizations of Datalog

Keywords :

formal methods, proof assistants, logic, databases

In many areas of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to help programmers create reliable systems. For example, in processor development, automated theorem proving uncovers deep bugs in designs before they become costly errors in silicon ; avionics developers use program analysis to verify critical safety properties of the embedded software running on airplanes ; and operating system vendors have successfully used model checking to eliminate entire classes of bugs in device drivers. But, until recently, data intensive management systems have largely resisted analysis using formal techniques.

However, current data management applications and systems involve increasingly massive data volumes. These are precious and their availability, integrity and reliability is a gold mine for companies, scientists and simply citizens. Since it is important to protect data integrity and reliability, one should ensure that the systems managing such data are indeed safe and reliable. Obtaining strong guarantees requires the use of formal methods and mature tools. A very promising approach consists in using proof assistants.

Program verification and certification have been intensively studied in the last decades yielding very impressive results and highly reliable software. Surprisingly, while the amount of data stored and managed by data engines has drastically increased, little attention has been devoted to ensure that such complex systems are indeed reliable. Among them, relational database management systems are the most widely spread. Also, deductive systems represent an unifying framework for a multitude of query systems and are the object of a resurge of interest in the context of the Semantic Web. Consequently, the observations motivate our choice to focus on the formalization of the relational data model, as well as on the deductive one - restricted to the Datalog fragment.

This thesis presents a Coq formalization of fundamental database languages and algorithms. It provides formal specifications stemming from two different approaches in defining database models : relational and logic based.

As such, the first contribution of the thesis is the development of a Coq library for the relational model. This contains formalizations of the relational algebra and of conjunctive queries. It also includes a mechanization of integrity constraints and of their inference procedures. We model two types of dependencies, which are among the most widely used : the functional dependencies and the multivalued dependencies, as well as their corresponding axiomatizations. We formally prove the soundness of their inference algorithms and, for the case of functional dependencies, also the completeness. These types of dependencies are instances of more general ones, namely general dependencies : equality generating dependencies and, respectively, tuple generating dependencies. We model general dependencies and their inference procedure, i.e, ”the chase”, for which we establish the soundness. Finally, we formally prove the fundamental database theorems, i.e, algebraic equivalence, the homomorphism theorem and the minimization of conjunctive queries.

A second contribution consists of the development of a Coq/ssreflect library for logic programming, restricted to Datalog. As part of this work, we provide a first mechanization of a standard Datalog engine, as well as of its extension with negation. The library contains a formalization of their model-theoretic semantics, together with the fixpoint one, implemented through a stratified evaluation procedure. The library is complete with the corresponding soundness, termination and completeness proofs. In this setting, we construct a preliminary framework for reasoning about stratified programs. The platform paves the way towards the certification of data-centric applications.

Contents

Acknowledgments

I am indebted to my thesis advisors, Véronique Benzaken and Évelyne Contejean, for the insight with which they have led me in exploring this immensely fruitful topic. As a result, our collaboration was truly a unique opportunity for me to progress in a variety of challenging research areas. This would not have been possible without the technical expertise they have shared with me, both on the database and the theorem proving sides. On a personal note, I am very grateful for their unfaltering patience, support, and belief in my ability to become an independent researcher. Thank you!

I am grateful to Sandrine Blazy and Mohand-Saïd Hacid for having accepted to review my thesis and for the feedback they have provided. I also thank Sarah Cohen-Boulakia and Michaël Rusinowitch for being part of my defense jury.

I wish to also thank all the members of VALS team. It was a pleasure to work in such a stimulating and friendly environment. In particular, I wish to thank Christine Paulin, Sylvain Conchon, and Thibaut Balabonski, whose classes I had the pleasure to teach. I am also thankful to all my teaching colleagues at the Science Faculty of Université Paris-Sud and the IUT d'Orsay.

A big thank you to all my past office mates: Paolo, Catherine, David, and Hai! You brightened my days, empathized with me in times of setback, encouraged me to persevere and were a pleasure to be around.

I am thankful to Michael Kohlhase and Florian Rabe for having been the first to encourage me to pursue research in computer science. The time spent in the KWARC group has been crucially formative for my career. I wish to also thank my internship supervisors from DFKI, the logic group at the Max Plank Institute and the Marelle team at Inria Sophia-Antipolis.

I was lucky enough to have fantastic roommates throughout my postgraduate studies, at Bourg-la-Reine (thank you, Alexandra!) and at Ivry-sur-Seine (thank you, Clara, Balthazar, Mathieu, Sonia and Adeline!). I would not have made it this far without the love that my friends have shown me time and again these past years. Thank you (in no particular order), Monica, Isabelle, Aivaras, Madalina, Cody, Rachel, Anca, Andjela, Clement, Elena, Georgi, Emilio, Aladin, Jean-Baptiste, Alin, Thomas, Allekta, Asma, Luana, Capucine. Thank you, Gabriel, Irène, Anne and Denis!

I thank my parents for always being there for me. I owe you everything.

Part I.

Introduction

1. Preliminaries

1.1. Motivation

This thesis describes a foundational formalization effort bridging *formal verification* and *database theory*. In the present section, we overview the recent developments, stemming from both areas, that motivated our endeavour.

Computers have become ubiquitous, handling *increasingly large* amounts of *sensitive* data. Consequently, we are, more than ever, reliant on their *correct* functioning. However, many malfunctions have historically led to economic and human catastrophes, from airplane, space probe and rocket explosions, to failed missile interceptions, to trading disruptions and fatal medical machinary glitches. This need for *strong guarantees* has ushered in the use of *formal methods*, which apply mathematical and logical tools to the design and implementation of software. Not only have these methods been integrated in the official standard for safety-critical systems, but their indisputable role in the development of cost-effective and low-defect products has made them gain ground as mainstream. Indeed, their application domains have expanded, both in research and in industry. These now range from language semantics, compilers, operating systems and security protocols - to embedded systems in the railway, automotive, nuclear, aeronautic and aerospatial settings and commercial processors. Moreover, the significant progress and refinement of such methods has even led to them assisting mathematical reasoning, as demonstrated by the development of large machine-checked proofs.

Formal methods combine *modeling* and *analysis* techniques, aimed at the *specification*, *implementation* and *verification* of languages and systems. *Formal specifications* unambiguously define the desired features of a given language, i.e, its syntax and semantics, or of a system's underlying algorithms, i.e, their expected behaviour. Such stringent descriptions can serve as design guidelines and - in some cases - as direct basis for a *formal implementation*, resulting in correct-by-construction code. *Formal verification* aims to statically ensure that characteristic properties, e.g., soundness, termination and completeness, are preserved dynamically, i.e, at execution.

Due to the *undecidability* of exhaustively finding potential run-time flaws through formal static analysis, a wide variety of approximate solutions have been developed. Depending on their nature and scope, these can be *roughly* classified into two main complementary approaches: *proof-based* (i.e, deductive verification, whose implementation tools rely on automatic theorem provers and semi-automatic proof assistants) and *exploration-based* (i.e, model checking, whose implementation tools rely - among others - on symbolic

and bounded model checking algorithms). While each have advantages and drawbacks, proof-based techniques were particularly well-tailored to our purposes, because of their expressivity and scalability to more complex systems.

The Coq proof assistant distiguishes itself among existing proof tools, through its support of high-order logic, proof script automation and code extraction, as well as through its successful use in a large number of areas. Prominent projects based on Coq led to **certified compilers and operating systems**, e.g, the CompCert verified C compiler ([16], [17], [53], [54]), the Vellvm verified LLVM interpreter ([85]) and the CertiKOS verified kernel for cloud computing ([44]), of **formalized programming frameworks**, e.g, the YNOT environment for imperative programs ([64]) and the Certicrypt environment for cryptography proofs ([11]), of **mechanized programming language semantics**, e.g, the JSCert formal JavaScript semantics ([18]) and the CakeML implementation of ML ([51]), of **certified termination tools**, i.e the CiME/Coccinelle library ([26]), and of **formalized mathematical proofs**, e.g, the Four Color theorem ([37]) and the Feit-Thompson (odd-order) theorem ([39]).

These results have paved the way for envisioning a transfer of the methodologies underlying proof assistant usage to the field of software engineering at large. As such, the *Deep Specification Expedition in Computing* project marks the emergence of what is called the *science of deep specification*. This new paradigm targets the principled development of real-world systems, based on full formalizations of their specification.

The DataCert project positions itself in the frame of this general setting and aims at the *deep specification* of data-centric systems. Part of its objectives are to thus provide mechanizations of the theoretical fundaments underlying, on the one hand, relational database management systems and, on the other hand, data integration and exchange systems. In accordance with some of these goals, we direct our *foundational* formalization efforts towards *relational* and *deductive* databases. The key motivations are as follows.

- **Relational databases** have long been a *prevalent* and popular storage solution, in particular for systems handling *safety-critical data*, such as security credentials, sensitive user information, financial, medical and legal records. Consequently, obtaining *strong safety guarantees* with respect to confidentiality and integrity has become crucial.
- **Deductive databases**, while much less used in practice than their relational counterpart, have a *simpler*, purely *declarative* and more *expressive* formalism. At a theoretical level, this provides a generic, unifying framework for a plethora of application-specific query languages. Moreover, its main exponent – Datalog – has been the object of a recent resurge in interest, as described in [9], and found many new applications in semantic web and ontology reasoning, data integration and exchange, security, networking, cloud computing, program analysis, etc. An overview is given in [48].

1.2. State of the Art

1.2.1. Formalizations in the Relational Database Setting

First efforts in this area were carried out by [41], in the Agda proof system. This work investigates different formalizations of the *unnamed* relational model, focusing on data definition and relational algebra aspects. More recently, a formalization by [60] provides a fully verified, lightweight *implementation* of a single-user relational database system that is also based on the *unnamed* relational model. The authors prove that their implementation meets formal requirements, all the proofs being written and verified in the YNOT extension of Coq (see [22]). Compared to these works, we consider the *named* version of the relational model, as it is the one implemented in real systems, such as Oracle, DB2, PostgreSQL or Microsoft Access. Also, we additionally cover conjunctive queries, optimization techniques and integrity constraint aspects.

1.2.2. Formalizations in the Deductive Database Setting

The work of [49] provides a Coq formalization of the correctness and equivalence of forward and backward, top-down and bottom-up semantics, based on a higher-order abstract syntax for Prolog. Related to the formalizations we describe in Section 8 in that it provides - among others - formal soundness proofs regarding fixpoint semantics, it differs from our work, as it follows an abstract interpretation perspective. Also, while we do not consider, on the syntax side, function symbols and, on the semantics side, forward and backward and top-down approaches, we do support negation and manage to establish correctness, as well as completeness properties of the underlying algorithms that bottom-up inference engines employ.

The work presented in [83][1] gives a Coq mechanization of standard Datalog, in the context of expressing distributed security policies. The development contains the encoding of the language, of bottom-up evaluation and the proofs corresponding to decidability. In our corresponding Coq/SSReflect formalizations, we did not need to explicitly need to prove decidability, as we took care to carefully set up our base type, such that this property is inferred automatically (see Chapter 8). While we did not also take into account the part concerning the modeling of security policy aspects, the scope of the results we proved is wider by comparison.

1.3. Problem Statement

Many technologies and algorithms, routinely and widely used in the database world, are not formalized or are, very often, underspecified. Indeed, we are on the threshold of developing the machinary required for them to incorporate the strong guarantees Coq certification provides. In light of reasons detailed in Section 1.1, we see this as a

[1] `http://www.cs.nott.ac.uk/types06/slides/NathanWhitehead.pdf`

shortcoming that we seek to address. To this end, we consider it a crucial and necessary step to *formally model* the foundational blocks on which they were built as deep specifications. We aim to mechanize underlying models and logical inference engines from the relational and deductive database setting, in an exploratory proof-engineering endeavour, focused on component scalability and reusability.

1.4. Thesis Contributions

The technical contributions of this thesis are subsumed by work in two formal developments: first, a Coq library for the relational model and, second, a ssreflect library for logic programming in the Datalog fragment. We detail them as follows.

- a formalized relational model library described in [14]
  - *formalization of the data model* (relations, tuples, etc.)
  - *mechanization of integrity constraints*:
    we encode general dependencies and prove soundness of their inference procedure; also, we encode instances corresponding to the two general dependency subclasses: functional and multivalued dependencies; we establish soundness of their inference, as well as completeness in the case of functional dependencies
  - *mechanization of the main relational query languages*:
    we encode relational algebra and a restricted fragment of relational calculus, i.e, conjunctive queries
  - *proof of the main "database theorems"*:
    algebraic equivalences, the homomorphism theorem and conjunctive query minimization
- a formalized inference engine for positive Datalog programs
  - *using the finite machinary of* ssreflect *in the logic programming setting*:
    we greatly simplify the verification effort, by assuming finiteness in the core thesis developments, without loss of generality; indeed, as it is well-known, every Datalog program has a finite model, thus working in the finite setting suffices. We perform this reduction separately from the main development, which, in our opinion, allows for a clearer presentation.
  - *a scalable mechanization of the syntax and semantics of positive Datalog*
  - *mechanization of the bottom-up evaluation heuristic*:
    formalization of an iterative monadic matching algorithm for terms, atoms and clause bodies, with corresponding soundness and completeness proofs
  - *formal characterization of the positive engine*:
    the key properties we establish are soundness, termination, completeness and model minimality, based on proofs we give for monotonicity, boundedness and stability, together with fixpoint theory results

- a formalized inference engine for DATALOG programs with negation
  - *mechanization of the syntax and semantics of Datalog programs with negation*
  - *mechanization of stratified evaluation*:
    we formalize program stratification and slicing of programs and interpretations; in order to reuse the positive engine, we translate negated literals to flagged positive atoms and extend the notion of an interpretation to that of a "complemented interpretation";
  - *extending the theory of the positive engine*:
    a crucial part of the stratified evaluation relies on the positive engine to perform evaluation of negative programs encoded as positive; this reuse requires an extension to the theory of the positive engine with incrementality and modularity lemmas
  - *formal characterization of the negative engine*:
    the key properties we establish are soundness, termination and completeness and model minimality

2. Methodology

In constructing formal specifications of database models and algorithms, as mentioned in the previous chapter, we follow a methodology based on *theorem proving* techniques. These are aimed at building and verifying theorems using software, whose trusted kernel is based on a specific set of inference rules. This concept can be traced back to the Automath project [29] and to Milner's *Logic of Computable Functions* (LCF) architecture [62], whose kernel implements natural deduction (see Section 3.4.2). Theorem provers can be automatic or semi-automatic (interactive), i.e, requiring a user to work in tandem with the computer. We focus on the latter kind, whose specification languages are based on more expressive logics, thus rendering their application scope much larger. In this setting, a typical workflow consists of providing an input program, together with its specification, and helping the prover construct a (mechanized) proof that the program's output complies with the specification. Consequently, the problem of ensuring program soundness is reduced to that of ensuring soundness of the underlying specification. Following this approach, we developed the formalizations presented in Chapter 7 and, respectively, in Chapters 8 and 9, in the Coq proof assistant and its ssreflect extension. In this chapter we briefly review these tools and libraries, highlighting the main features we exploited.

2.1. Coq

The Coq proof assistant ([61]) belongs to the class of interactive theorem provers, alongside systems from the HOL family, e.g, Isabelle/HOL ([65]), HOL-light ([45]), PVS ([68]), - in the classic higher-order logic tradition - and alongside systems such as NuPrl ([25]), Agda ([66]) and Matita ([6]) - in the constructive type theory tradition. Its core formalism is the Calculus of Inductive Constructions (CIC) [70], which corresponds to the most expressive type theory in Barendregt's lambda cube [10], namely that supporting polymorphism, type operators and dependent types. The Coq system and its underlying theory are described in [15].

The Calculus of Inductive Constructions is based on the Curry-Howard "propositions-as-types and proofs-as-programs" paradigm ([27], [47]). As such, Coq is both a proof system and a functional programming language (Gallina). Essentially, the type system can be seen as containing only function types and inductive types. Coq's equality is *intensional* and type checking is decidable. The system in implemented in Objective Caml, a dialect of ML, and comes with an automatic extraction mechanism from Coq proof specifications that can be used to build certified and efficient functional programs. Proving a theorem T in Coq involves finding a program p, such that $\vdash p : T$, i.e, that

p inhabits the type T. Alternatively, it involves constructing a typing derivation that assigns the type T to the program p.

A classical example of a proposition is the "and" datatype with a single constructor, encapsulating two proofs, usually written as $\wedge$. To prove $A \wedge B$, one needs to find a proof a for A and a proof b for B and then apply the conjunction constructor. A more interesting datatype is $\vee$. This datatype has two constructors, indicating whether one has found a proof either for A or for B. It is easy to see that to provide a proof for the excluded middle principle, i.e, $A \vee \neg A$, one would have to find a proof for either A or $\neg A$. This is unprovable in general, as there exist propositions that cannot be decided in this way. For example, if A expresses Turing machine termination, the principle is equivalent to the halting problem. Similarly, to prove $\exists x, Px$, one has to provide a witness w and a proof of Pw. Consequently, $\neg(\forall x, \neg Px)$ does not imply $\exists x, Px$, as this is equivalent to the excluded middle.

As building programs by hand is cumbersome, Coq provides a set of so called *tacticals*, for building Gallina programs. We will discuss tacticals more in the following section.

2.2. Ssreflect

The model-theoretic semantics of Datalog programs is deeply rooted in finite model theory. To quote [57]:

> "For many years, finite model theory was viewed as the backbone of database theory, and database theory in turn supplied finite model theory with key motivations and problems. By now, finite model theory has built a large arsenal of tools that can easily be used by database theoreticians without going to the basics (...) "

Given this foundational bias, we chose to rely, in our formalizations from Chapter 8 and Chapter 9, on the Mathematical Components library[1], built using the Ssreflect ([40]) extension of Coq. This library is especially well-suited for our purposes, as it was the basis of extensive formalizations of finite model theory, in the context of proving the Feit-Thompson theorem ([39]), central to finite group classification.

2.2.1. Ssreflect Tactics

A comprehensive presentation of Ssreflect functionalities is in [40]. We briefly overview the most important tactics next. Generally, the tactics ssrflect provides can be split in two types: the "defective" tactics, i.e, `move`, `case`, `have`, `elim` and `apply`, which behave similarly to their Coq counterparts, and the rewrite tractics, comprising unfolding, simplification and rewrite. Tactics are additionally constructed combining the "defective" ones with context-manipulation "tacticals", i.e, `:` and $\Rightarrow$. The former moves facts and

[1] `http://math-comp.github.io/math-comp/`

constants to the goal, while the latter introduces variables, local definitions and assumptions to the context. SSREFLECT supports pattern selection mechanisms common to all tactics.

The defective `move` tactic exposes the first assumption in goals, i.e, changing $\sim$ `P` to `P` $\rightarrow$ `False`. The defective `case` tactic does case analysis on (co)inductive types, by deconstructing them, exposing their constructors and arguments and instantiating the latter correspondingly; it also does injection, if it finds an equality. The defective `elim` tactic performs inductive elimination on inductive types. Finally, backwards chaining reasoning is implemented via `apply`. A convenient feature is that these tactics can be chained with views, allowing to change boolean equality goals into equivalent propositions. We illustrate their use, by giving a step by step account of the proof below.

```
Lemma fwd_chain_cat def p1 p2 i :
  fwd_chain def (p1 ++ p2) i =
  fwd_chain def p1 i ∪ fwd_chain def p2 i.
Proof.
by apply/setP⇒ ga; rewrite ?(big_cat, inE); rewrite orbACA orbb.
Qed.
```

For the purposes of stating the `fwd_chain_cat` lemma, let us assume a default constant `def`, an initial interpretation (finite set of ground atoms) `i`, and the (positive Datalog) programs `p1` and `p2`. We prove that applying forward chain to the concatenation of `p1` and `p2` equals the union of applying forward chain to `p1` and to `p2`.

To this end, we use a crucial property of finite sets, namely that set equality is equivalent to the extensional one. This is expressed by the `setP` lemma from the `finset` library. Note that, for any finite sets, `A` and `B`, the `=i`[2] operator stands for $\forall$ `x, x` $\in$ `A` $=$ `x` $\in$ `B`.

```
setP : ∀ (T : finType) (A B : {set T}), A =i B ↔ A = B,
```

Transforming the goal with the `setP` view, naming and moving the universally quantified arbitrary set element `ga` into the context, the proof obligation becomes:

```
(ga ∈ fwd_chain def (p1 ++ p2) i) =
(ga ∈ fwd_chain def p1 i ∪ fwd_chain def p2 i).
```

Since for an arbitrary program `p`, we defined the forward chain function as:

```
fwd_chain def p i = i ∪ \bigcup_(cl ← p) cons_clause def cl i
```

the goal is equivalent to:

```
(ga ∈ i ∪ \bigcup_(cl ← (p1 ++ p2)) cons_clause def cl i) =
    (ga ∈ i ∪ \bigcup_(cl ← p1) cons_clause def cl i ∪
      (i ∪ \bigcup_(cl ← p2) cons_clause def cl i)).
```

[2]Note that the `i` in `=i` is different from the interpretation parameter `i` in the `fwd_chain_cat` lemma

Next, we optionally rewrite all further subgoals with a rule tuple containing the generic `inE` membership simplification and the `big_cat` rule below, for decomposing big operations over program concatenations:

```
                \big[op/idx]_(j ← (p1 ++ p2) | P j) F j =
op (\big[op/idx]_(j ← p1 | P j) F j) (\big[op/idx]_(j ← p2 | P j) F j)
```

Instantiating the latter, by replacing `op` with the union operation, we obtain:

```
                \bigcup_(j ← (p1 ++ p2) | P j) F j =
      (\bigcup_(j ← p1 | P j) F j) ∪ (\bigcup_(j ← p2 | P j) F j)
```

The goal is thus transformed into an equality of boolean disjunctions.

```
ga ∈ i || ga ∈ (\bigcup_(cl ← p1) cons_clause def cl i) ||
      ga ∈ (\bigcup_(cl ← p2) cons_clause def cl i) =
ga ∈ i || ga ∈ (\bigcup_(cl ← p1) cons_clause def cl i) ||
  ga ∈ i || ga ∈ (\bigcup_(cl ← p2) cons_clause def cl i)
```

This is solvable by rewriting with the boolean interchange lemma `orbACA` and with the boolean idempotency lemma `orbb`, from the `ssrbool` library.[3]

```
Lemma orbACA (b1 b2 b3 b4 : bool) :
  (b1 || b2) || (b3 || b4) = (b1 || b3) || (b2 || b4).
Lemma orbb (b : bool) : b || b = b.
```

Finally, we chain all tactics using `;` and prefix the proof with the `by` terminator, to explicitely discharge trivial subgoals.

Next, we illustrate the `elim` tactic with the proof for the `iter_fwd_chain_sym` lemma.

```
Lemma iter_fwd_chain_sym def p i ga k :
  (ga ∈ iter k (fwd_chain def p) i) →
  (ga ∈ i) || (sym_gatom ga ∈ hsym_prog p)
```

This establishes that, given an initial interpretation `i`, a ground atom `ga` is in the result of `k` iterations of the forward chain procedure on a program `p`, either if `ga` is in `i` or if the symbol of `ga` is among the head clause symbols in `p`. The proof is by straightforward induction on `k`. Note the pattern `[→ | k h]` after the introduction tactical, corresponding to the base and step case branches. For the first, we can rewrite and close the goal with the top sequent. For the second, we aim to prove:

```
(ga ∈ fwd_chain def p (iter k (fwd_chain def p) i) →
      (ga ∈ i) || (sym_gatom ga ∈ hsym_prog p)
```

[3] Alternatively, this is also provable by case analysis on `(ga ∈ i)`.

We know that the following lemma holds:

```
Lemma fwd_chain_sym def p i ga :
  (ga ∈ fwd_chain def p i) →
  (ga ∈ i) || (sym_gatom ga ∈ hsym_prog p).
```

Hence, transforming the goal with `fwd_chain_sym`, we obtain:

```
(ga ∈ iter k (fwd_chain def p) i) || (sym_gatom ga ∈ hsym_prog p) →
                (ga ∈ i) || (sym_gatom ga ∈ hsym_prog p)
```

Applying the reflection lemma `orP`:

```
Lemma orP (b1 b2 : bool) : reflect (b1 ∨ b2) (b1 || b2).
```

the goal is split into two subgoals, for each disjunct. The first is provable directly from the induction hypothesis:

```
ga ∈ iter k (fwd_chain def p) i →
(ga ∈ i) || (sym_gatom ga ∈ hsym_prog p).
```

The second is provable by rewriting with the top sequent → and with the boolean lemma:

```
Lemma orbT (b : bool) : b || true.
```

The complete proof for `iter_fwd_chain_sym` is thus:

```
Lemma iter_fwd_chain_sym def p i ga k :
  (ga ∈ iter k (fwd_chain def p) i) →
  (ga ∈ i) || (sym_gatom ga ∈ hsym_prog p).
Proof.
by elim: k ⇒ [→|k h] //=/fwd_chain_sym/orP [/h|→]; rewrite ?orbT.
Qed.
```

2.2.2. Library Overview

The Mathematical Components library provides a development ranging from basic data structures and types to linear algebra, Galois and finite group theory. In particular, the support for finite types is mature, as it lies at the basis of many other developments. The library is organized in an object-oriented way, with types implementing interfaces and mixins implementing interface inheritance. The low level implementation of the class system is done by the canonical structures mechanism, allowing users to supply solutions to particular unification problems involving records; this is detailed in [35]. Next, we quickly present the main classes and libraries we used in our development.

ssrfun The file contains the main lemmas and definitions about functions. For instance, lemmas about injectivity.

ssrbool The Mathematical Components library is highly specialized in dealing with *decidable predicates*. This file develops their corresponding base theory.
Decidable predicates are predicates whose truth can be computationally determined or, in Coq terms, `P : Prop`, such that `{P} + {not P}` is provable [4]. It is often convenient to represent the truth of `P` as a concrete algorithm `p : bool`. Indeed, `reflect P p` does exactly that, i.e, `reflect P true` is provable only when `P` is provable and `reflect P false` is provable only when `not P` is provable.
The `ssrbool` file also provides support for *generic predicates*, e.g, the `\in` predicate, widely used in the library to implement membership comprehensively, allowing one to, for instance, write `x \in l = x \in s`, where `l` is a list and `s` is a set.

eqtype The file provides the base class of the Mathematical Components hierarchy: the `eqType` class of types with decidable equality, written `x == y`. The main corresponding lemma is `eqP`, which states that the computational equality corresponds to the propositional one, i.e `(x = y) <-> (x == y)`. Also, this file contains the `subType` class, which provides generic support for well-behaved sigma types of the form `{x | P x}`, where `P` is a boolean predicate. A subtype coerces to the base type automatically, and thanks to the boolean nature of `P`, the proof is irrelevant, allowing us to transfer all the properties of the base type to the subtype.

ssrnat The file provides a theory of the natural numbers, with the more covenient choice of boolean operators for orders, e.g, given two natural numbers `m` and `n`, the less or equal comparison `leq` is defined as computationally checking `m - n == 0`.

seq The file provides a theory of lists or sequences, with generic predicate membership, i.e, `(x \in l)`, if `x` is an `eqType`.

fintype The file is crucial to our development, as it provides the `finType` class of finite types. Finiteness is defined as the duplicate-free enumeration of elements of a given type, i.e, `∀x : T, count x (enum T) = 1`. The library also provides the type of ordinals `'I_n = { m : nat | m < n }`. Moreover, we take advantage of the `finType` instance for tuples, which provides a convenient way of encoding, for example, a program's Herbrand base, given the maximal arity `n` of its predicates and its active domain `adom`, i.e, `enum (n.-tuple adom)`. Another interesting property supported by the library is that quantification becomes decidable. For this purpose, the following constructs are supported: `[∀x, P x]` and `[∃x, P x]`, together with the reflection lemmas `forallP` and `existsP`, relating the computational definitions of quantification to their usual counterparts. This is very handy, as, in this setting, classical logic principles hold, e.g, `~ ~[∀x , ~ ~ P x] = [∃x, P x]`.

[4] The "+" operator is the computational version of "or".

finfun The file provides a theory of finitely-supported functions. Given a finite type, functions from it are represented extensionally as their graph, i.e the list of values the function assigns to every domain element.

finset The file provides a theory of sets over finite types. Given an underlying finite type, these are represented as lists of booleans, corresponding to the set membership or characteristic function. The main property is set extensionality (`setP`), i.e, `(forall x, x \in s1 = x \in s2) <-> s1 = s2`, which is a very convenient property for reasoning about sets point-wise.

bigop We also take advantage of the big operator library (see [15]), which adds monoid classes to the Mathematical Components hierarchy. This library provides us with indexed operators of the form:

```
\bigop[op/zero]_l (fun x ⇒ P x).
```

In particular, by instantiating the operator `op` and the lower limit `zero` with `setU` and, respectively, `set0`, we obtain set union and, by instantiating them with `addn` and `0`, we obtain the big sum.

2.3. Thesis Outline

The thesis contents are divided into a Theoretical Overview and a Formalization Overview. In the first part, we start with Section 3, by reviewing basic first-order logic definitions related to its syntax, semantics and inference systems. In Section 4, we review aspects from relational model theory that were relevant to our work: we discuss data representation in Section 4.1, data extraction in Section 4.2 and data integrity in Section 4.3. In Section 5, we present the Datalog language, namely its syntax, in Section 5.1, and its minimal model and fixpoint semantics, in Section 5.2.1 and Section 5.2.2. Finally, in Section 6, we present the extension of Datalog with negation, namely its syntax in Section 6.1 and its stratified semantics, in Section 6.2. In the second part, we present corresponding formalizations of the relational model, in Section 7, and of two Datalog engines: a standard one, in Section 8, and one extended with negation, in Section 9. We give perspectives of our work and draw conclusions in Section 10 and Section 11.

Part II.

Theoretical Overview

3. First-Order Logic

The logic roots of database inference became apparent in the context of converging database and artificial intelligence research, following early advances in automated theorem proving. As overviewed in [63], [50], [56] and [42] noted the relevance of the resolution principle [75] to question answering systems. In [23] a logic based query language, i.e, the relational calculus, is introduced. The work by [81] laid out the foundations of logic programming, spurring further interest in applying logic to databases. In [74], a logical reconstruction of database theory is proposed, revealing foundational assumptions, i.e, closed-world, unique-name and domain-closure. The field of deductive databases soon developed, culminating with DATALOG as its prominent language. The formalizations of the relational model (Chapter 7) and of DATALOG (Chapter 8 and Chapter 9) are based on the theory of first-order logic. We recall basic concepts related to its *syntax* and *semantics*, following [28].

3.1. Syntax

The base syntactical building blocks of first-order languages are *signatures*. These fix the non-logical symbols of the language (its vocabulary) and are made explicit in our mechanizations, as type declarations.

Definition 3.1.1 (Signatures)**.** *A first-order* signature Σ *is a triple* $(\mathcal{F}, \mathcal{P}, ar)$ *where*

- $\mathcal{F}$ *and* $\mathcal{P}$ *are pairwise disjoint sets of function and predicate symbols*
- $ar : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ *is an arity mapping*

We sometimes just write $\Sigma \equiv (\mathcal{F}, \mathcal{P})$ and denote with $f_{/n}$ and $p_{/m}$ functions and predicate symbols, such that $ar(f) = n$ and $ar(p) = m$. Also, note that:

- If $ar(f) = 0$, for $f \in \mathcal{F}$, f is called a *constant*.
  We denote the set of constants with $\mathcal{C}$, where $\mathcal{C} \subseteq \mathcal{F}$.
- If $ar(p) = 0$, for $p \in \mathcal{P}$, p is called a *propositional variable*.

Examples of such structures are Peano arithmetic ($\mathcal{F} = \{0_{/0}, s_{/1}, +_{/2}, \times_{/2}\}, \mathcal{P} = \emptyset$), groups ($\mathcal{F} = \{e_{/0}, inv_{/1}, \circ_{/2}\}, \mathcal{P} = \emptyset$)) and lattices ($\mathcal{F} = \emptyset, \mathcal{P} = \{\leq\}$).

Fixing a signature $\Sigma \equiv (\mathcal{F}, \mathcal{P})$, we can construct a first-order language $\mathcal{L}$, by further adding logical symbols, i.e, a countable set of variables $\mathcal{X}$, quantifiers $(\forall, \exists)$ and connectives $(\wedge, \vee, \neq, \Rightarrow, \Leftrightarrow)$, paranthesis and punctuation symbols. The primitives of the language, i.e, its *words*, are called *terms* and are defined inductively below.

Definition 3.1.2 (Terms). *The set of $\mathcal{L}$-terms is the minimal set $T_\Sigma(\mathcal{X})$ satisfying*

- $\mathcal{X} \subseteq T_\Sigma(\mathcal{X})$ *and* $\mathcal{C} \subseteq T_\Sigma(\mathcal{X})$
- *for all* $f \in \mathcal{F}$, *if* $t_1, \ldots, t_{ar(f)} \in T_\Sigma(\mathcal{X})$ *then* $f(t_1, \ldots, t_{ar(f)}) \in T_\Sigma(\mathcal{X})$

Next, we can express base sentences, called *atomic formulas*, as sentential letters - $\bot$ (true) and $\top$ (false) - and as *atoms* (applications of a predicate symbol to a number of terms, as indicated by its arity). These can then be combined into more complex sentences, as captured by the following inductive *formula* definition.

Definition 3.1.3 (Formulas). *The set of $\mathcal{L}$-formulas is the minimal set $F_\Sigma(\mathcal{X})$ satisfying*

- $\bot, \top,\ p(t_1, \ldots, t_{ar(p)}) \in F_\Sigma(\mathcal{X})$, *where* $p \in \mathcal{P}$ *and* $t_1, \ldots, t_{ar(p)} \in T_\Sigma(\mathcal{X})$
- *if* $\phi_1, \phi_2 \in F_\Sigma(\mathcal{X})$ *then* $\phi_1 \square \phi_2 \in F_\Sigma(\mathcal{X})$, *where* $\square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- *if* $\phi \in F_\Sigma(\mathcal{X})$ *then* $\neg\phi \in F_\Sigma(\mathcal{X})$ *and* $(\forall x)\phi \in F_\Sigma(\mathcal{X})$ *and* $(\exists x)\phi \in F_\Sigma(\mathcal{X})$

Note that by viewing term equality as a predicate, i.e $= (t_1, t_2) \equiv t_1 = t_2$, we subsume the case : if $t_1, t_2 \in T_\Sigma(\mathcal{X})$ then $t_1 = t_2 \in F_\Sigma(\mathcal{X})$.

To summarize, a first-order language $\mathcal{L}$ is described by the grammar:

$$\begin{aligned} \text{Terms } t &::= x,\ x \in \mathcal{X} \mid c,\ c \in \mathcal{C} \mid f(t_1, \ldots, t_n),\ f \in \mathcal{F},\ ar(f) = n \\ \text{Atomic Formulas } A &::= \bot \mid \top \mid p(t_1, \ldots, t_n),\ p \in \mathcal{P},\ ar(p) = n \\ \text{Complex Formulas } \phi, \psi &::= A \mid \phi \square \psi,\ \square \in \{\wedge, \vee, \Rightarrow\} \mid \neg\phi \mid (\forall x)\phi \mid (\exists x)\phi \end{aligned}$$

In Chapters 5, 6, 8 and 9, we will focus on the DATALOG language, a first-order language *without function symbols*. Its formulas are of a particular kind, i.e, *clausal*. Namely, these are disjunctions of positive or negated atomic formulas (literals), i.e, $C ::= L_1 \vee \ldots \vee L_n$, where $L ::= A \mid \neg A$.

It is routinely the case that first-order logic sentences have to be syntactically transformed to ease their manipulation. Such transformations, i.e *substitutions*, operate on contexts, consisting of sets of variables. Depending on whether or not a variable is quantified, a distinction is made between *bound* and *free* variables. Let us denote with BV and FV the sets of bound and, respectively, free variables of a term or formula.

Definition 3.1.4 (Term Variables). *The set $FV(t)$ of free variables occuring in an $\mathcal{L}$-term t is defined as :*

- $FV(x) = \{x\}$ *and* $FV(c) = \emptyset$
- $FV(f(t_1, \ldots, t_{ar(f)})) = \bigcup_{i=1}^{ar(f)} FV(t_i),\ t_i \in T_\Sigma(\mathcal{X})$

An $\mathcal{L}$-term t is *ground* (*closed*), if $FV(t) = \emptyset$. We denote ground terms set with T_Σ.

Definition 3.1.5 (Formula Variables)**.** *The set $VAR(\phi)$ of variables occuring in an $\mathcal{L}$-formula ϕ is defined as $VAR(\phi) = FV(\phi) \cup BV(\phi)$, where :*

- $FV(\bot) = BV(\bot) = \emptyset$ *and* $FV(\top) = BV(\top) = \emptyset$
- $FV(p(t_1, \ldots, t_{ar(p)})) = \bigcup_{i=1}^{ar(p)} FV(t_i)$, $t_i \in T_\Sigma(\mathcal{X})$ *and* $BV(p) = \emptyset$
- $V(\phi \square \psi) = V(\phi) \cup V(\psi)$, $V \in \{FV, BV\}$, $\square \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- $V(\neg\phi) = V(\phi)$, $V \in \{FV, BV\}$
- $FV((\forall x)\phi) = FV((\exists x)\phi) = FV(\phi) - \{x\}$
- $BV((\forall x)\phi) = BV((\exists x)\phi) = BV(\phi) \cup \{x\}$

An $\mathcal{L}$-formula ϕ is called a *sentence* or *closed* if $FV(\phi) = \emptyset$ and *ground* if $VAR(\phi) = \emptyset$. The set of $\mathcal{L}$-sentences is denoted as $SEN_\mathcal{L}$.

Definition 3.1.6 (Substitutions)**.** *A substitution σ is a mapping $\sigma : \mathcal{X} \to T_\Sigma(\mathcal{X})$. Extensionally, σ is represented as $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$, with $t_i \in T_\Sigma(\mathcal{X})$ and pairwise distinct $x_i \in \mathcal{X}$, for $i \in [1, n]$.*

Substitution application results in *instances* or *instantiations*, constructed as follows.

Definition 3.1.7 (Substitution Application)**.** *For a variable $x \in \mathcal{X}$, the instantiation of x with a substitution $\sigma = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$, denoted as σx, is :*

$$\sigma x = \begin{cases} t_i & x = x_i \\ x & \textit{otherwise} \end{cases}$$

The set $\{x \in \mathcal{X} \mid \sigma x \neq x\}$, called the domain *or* support *of σ, is abbreviated $dom(\sigma)$.*

Even though σ operates on variables, it can be extended to a mapping $\bar{\sigma}$ over terms $t \in T_\Sigma(\mathcal{X})$, replacing all $x \in VAR(t)$ with σx :

$$\sigma t = \begin{cases} t & t \in \mathcal{C} \\ f(\sigma t_1, \ldots, \sigma t_n) & t = f(t_1, \ldots, t_n) \end{cases}$$

Based on this, the instantiation of a formula ϕ can be defined as:

$$\sigma\phi = \begin{cases} \bot & \phi = \bot \\ p(\sigma t_1, \ldots, \sigma t_n) & \phi = p(t_1, \ldots, t_n) \\ \sigma\phi_1 \square \sigma\phi_2 & \phi = \phi_1 \square \phi_2, \square \in \{\wedge, \vee, \Rightarrow\} \\ \neg(\sigma\psi) & \phi = \neg\psi \\ (\square x)\sigma'\psi, \text{ where } \sigma' = \sigma \setminus [x \mapsto \sigma x] & \phi = (\square x)\psi, x \in dom(\sigma), \square \in \{\forall, \exists\} \\ (\square x)\sigma\psi & \phi = (\square x)\psi, x \notin dom(\sigma), \square \in \{\forall, \exists\} \end{cases}$$

A notion that does not usually appear in standard presentations, but that we exploit in Section 8.3.3, is that of a *substitution ordering*, defined below.

Definition 3.1.8 (Substitution Ordering)**.** *Let us restrict ourselves to only consider terms with no function symbols. Given substitutions σ_1 and σ_2, such that $dom(\sigma_1) \subset dom(\sigma_2)$, we define a* substitution ordering $\preceq$ *as :*

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

3.2. Semantics

Let us fix a signature $\Sigma = (\mathcal{F}, \mathcal{P}, ar)$ and a language $\mathcal{L}$ over Σ. To give meaning to $\mathcal{L}$, we start by fixing a domain of discourse or universe $U_{\mathfrak{M}}$. Based on this, we can assign denotations to all its non-logical constants. The most common way of doing so is by specifying *structures* over Σ.

Definition 3.2.1 (Σ-Structures [1])**.** *A* Σ-structure $\mathfrak{M} = (U_{\mathfrak{M}}, I)$ *consists of a* universe $U_{\mathfrak{M}} \neq \emptyset$ *and of an* interpretation function I*, such that:*

- *for every $c \in \mathcal{C}$: $c^I \in U_{\mathfrak{M}}$*
- *for every $f \in \mathcal{F}$, where $ar(f) = n$: $f^I : U^n_{\mathfrak{M}} \to U_{\mathfrak{M}}$*
- *for every $p \in \mathcal{P}$, where $ar(p) = n$: $p^I : U^n_{\mathfrak{M}} \to \{\top, \bot\}$*

In Chapters 5, 6, 8 and 9, we focus on *Herbrand* Σ*-Structures* $\mathcal{H} = (U_{\mathcal{H}}, I_{\mathcal{H}})$. These are based on the notion of a *Herbrand Universe* $U_{\mathcal{H}} = T_\Sigma$, consisting of the set of ground terms of the language. The corresponding *Herbrand Interpretation* $I_{\mathcal{H}}$ is defined in the standard way, i.e, $I_{\mathcal{H}} : \Sigma \to U_{\mathcal{H}} \cup \{\top, \bot\}$, such that:

- for every $c \in \mathcal{C}$: $c^{I_{\mathcal{H}}} = c \in U_{\mathcal{H}}$
- for every $f \in \mathcal{F}$, where $ar(f) = n$: $f^{I_{\mathcal{H}}} : U^n_{\mathcal{H}} \to U_{\mathcal{H}}$ is fixed as $f^{I_{\mathcal{H}}} = f$
- for every $p \in \mathcal{P}$, where $ar(p) = n$: $p^{I_{\mathcal{H}}} : U^n_{\mathcal{H}} \to \{\top, \bot\}$

Let us fix a Σ-structure, $\mathfrak{M} = (U_{\mathfrak{M}}, I)$. In order to evaluate the veracity of a formula, we need to associate an element of the domain of discourse to each variable. This is done by *valuations*, as defined below.

Definition 3.2.2 (Valuations [2])**.** *A* valuation ι *over* $\mathfrak{M}$ *is a mapping* $\iota : \mathcal{X} \to U_{\mathfrak{M}}$*. Given $x \in \mathcal{X}, u \in U_{\mathfrak{M}}$, we denote with $\iota[x \mapsto u]$, the* extension *of a valuation, such that, for a variable y, its application $\iota[x \mapsto u]y$ is defined as:*

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

Using valuations, we can interpret terms homomorphically and associate to each, an element from the domain of discourse.

[1] also called Σ-*interpretations* or Σ-*algebras*
[2] also called *variable assignments*

Definition 3.2.3 (Term Interpretation)**.** *The interpretation of $\mathcal{L}$-terms in $\mathfrak{M}$ under a valuation $\iota : \mathcal{X} \to U_{\mathfrak{M}}$ is given by a mapping $[\![\cdot]\!] : T_\Sigma(\mathcal{X}) \to U_{\mathfrak{M}}$, such that :*

$$[\![x]\!]^{I,\iota} = \iota x,\ [\![c]\!]^{I,\iota} = c^I \text{ and } [\![f(t_1, \ldots, t_n)]\!]^{I,\iota} = f^I([\![t_1]\!]^{I,\iota}, \ldots, [\![t_n]\!]^{I,\iota})$$

Having embedded all formula primitives into the universe $U_{\mathfrak{M}}$, we can inductively evaluate formulas as true or false, based on the definition below.

Definition 3.2.4 (Formula Evaluation)**.** *The evaluation of $\mathcal{L}$-formulas in $\mathfrak{M}$ under a valuation $\iota : \mathcal{X} \to U_{\mathfrak{M}}$ is given by the mapping $[\![\cdot]\!] : SEN_{\mathcal{L}} \to \{0, 1\}$, such that:*

- $[\![\bot]\!]^{I,\iota} = 0$ *and* $[\![p(t_1, \ldots, t_n)]\!]^{I,\iota} = \begin{cases} 1 & p^I([\![t_1]\!]^{I,\iota}, \ldots, [\![t_n]\!]^{I,\iota}) = \top \\ 0 & \text{otherwise} \end{cases}$
- $[\![\phi_1 \wedge \phi_2]\!]^{I,\iota} = \min([\![\phi_1]\!]^{I,\iota}, [\![\phi_2]\!]^{I,\iota})$ *and* $[\![\phi_1 \vee \phi_2]\!]^{I,\iota} = \max([\![\phi_1]\!]^{I,\iota}, [\![\phi_2]\!]^{I,\iota})$
- $[\![\phi_1 \Rightarrow \phi_2]\!]^{I,\iota} = \max(1 - [\![\phi_1]\!]^{I,\iota}, [\![\phi_2]\!]^{I,\iota})$ *and* $[\![\neg\phi]\!]^{I,\iota} = 1 - [\![\phi]\!]^{I,\iota}$
- $[\![(\forall x)\phi]\!]^{I,\iota} = \inf_{u \in U_{\mathfrak{M}}} \{[\![\phi]\!]^{I,\iota[x \mapsto u]}\}$ *and* $[\![(\exists x)\phi]\!]^{I,\iota} = \sup_{u \in U_{\mathfrak{M}}} \{[\![\phi]\!]^{I,\iota[x \mapsto u]}\}$

Note that, if $U_{\mathfrak{M}}$ is finite, we have:
$[\![(\forall x)\phi]\!]^{I,\iota} = \min_{u \in U_{\mathfrak{M}}} \{[\![\phi]\!]^{I,\iota[x \mapsto u]}\}$ *and* $[\![(\exists x)\phi]\!]^{I,\iota} = \max_{u \in U_{\mathfrak{M}}} \{[\![\phi]\!]^{I,\iota[x \mapsto u]}\}$.

The fundamental concepts in semantics are *satisfiability* and *validity*. These express if and under which conditions formulas are true.

Definition 3.2.5 (Formula Satisfiability)**.** *A formula $\phi \in F_\Sigma(\mathcal{X})$ is* satisfiable *iff there exists $\mathfrak{M} = (U_{\mathfrak{M}}, I)$ and $\iota : \mathcal{X} \to U_{\mathfrak{M}}$, such that $[\![\phi]\!]^{I,\iota} = 1$, denoted $\mathfrak{M}, \iota \models_I \phi$; otherwise, ϕ is* unsatisfiable*.*

We can thus define what it means for a formula to be true in general, based on its evaluation under a fixed structure.

Definition 3.2.6 (Formula Validity)**.** *A formula $\phi \in F_\Sigma(\mathcal{X})$ is* valid in $\mathfrak{M}$ *iff, for all $\iota : \mathcal{X} \to U_{\mathfrak{M}}$, $[\![\phi]\!]^{I,\iota} = 1$. This is denoted as $\mathfrak{M} \models_I \phi$ and $\mathfrak{M}$ is called a* model *of ϕ. The formula ϕ is* valid *in general iff, for all Σ-structures $\mathfrak{M}$, $\mathfrak{M} \models_I \phi$. This is denoted $\models \phi$.*

Based on this, we can introduce the central notion of *logical consequence*. This captures the relation between sets of sentences that are generally valid, as detailed next.

Definition 3.2.7 (Formula Entailment and Equivalence)**.** *A formula $\phi_1 \in F_\Sigma(\mathcal{X})$* entails *(or* implies*) a formula ϕ_2, where $\phi_2 \in F_\Sigma(\mathcal{X})$ is said to be a* semantic consequence *(or* logical implication*) of ϕ_1, denoted as $\phi_1 \models \phi_2$, iff, for all $\mathfrak{M} = (U_{\mathfrak{M}}, I)$ and $\iota : \mathcal{X} \to U_{\mathfrak{M}}$,*

$$[\![\phi_1]\!]^{I,\iota} = 1 \text{ implies } [\![\phi_1]\!]^{I,\iota} = 1, \text{ i.e, if } \mathfrak{M} \models_I \phi_1 \text{ then } \mathfrak{M} \models_I \phi_2$$

The formula ϕ_1 is equivalent *to ϕ_2, denoted as $\phi_1 \equiv \phi_2$, iff $\phi_1 \models \phi_2$ and $\phi_2 \models \phi_1$.*

For example, the following equivalences hold:

- $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$ and $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$
- $\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \vee \neg\phi_2$ and $\neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$ (De Morgan's laws)
- $\phi_1 \vee (\phi_2 \wedge \phi_3) \equiv (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$
- $\neg(\forall x\phi) \equiv \exists x\neg\phi$, $\neg(\exists x\phi) \equiv \forall x\neg\phi$ and $\neg\neg\phi \equiv \phi$
- $\phi \wedge \top \equiv \phi$, $\phi \wedge \bot \equiv \bot$, $\phi \vee \top \equiv \top$, $\phi \vee \bot \equiv \phi$

Given a signature and a structure interpreting it, we can define restrictions over the set of all possible sentences that can be expressed. These are called *theories* and consist of all ground sentences satisfying the signature's model structure.

Definition 3.2.8 (Σ-Theories)**.** *The first-order theory of a Σ-structure $\mathfrak{M} = (U_{\mathfrak{M}}, I)$ is defined as $Th(\mathfrak{M}) = \{\phi \in F_\Sigma(\mathcal{X}) \mid \mathfrak{M} \models_I \phi\}$*[3]*.*

Such theories occur naturally in logic, e.g, the theory of equality, as well as in mathematics, e.g the theory of Peano arithmetic, of Presburger arithmetic and those corresponding to integers, rationals and reals. In databases, query languages can be seen as first-order theories, as exemplified by the relational calculus (in Chapter 7) and by the DATALOG languages (in Chapters 8 and 6).

3.3. Normal Forms

Normal forms were introduced as a way of simplifying logical constructs, to obtain efficient structures for mechanized (theorem prover) reasoning. A first such transformation targets extracting all quantifiers to the head of a given formula.

Definition 3.3.1 (Prenex Normal Form)**.** *Any formula ϕ can be converted into an* equivalent *prenex formula $\square_1 x_1 \ldots \square_n x_n \psi$, $\square_i \in \{\forall, \exists\}$. The quantifier-free ψ is called* matrix *and $\square_1 x_1 \ldots \square_n x_n$ is called* quantifier prefix*. The transformation is denoted $\Rightarrow_P^*$.*

Next, skolemization replaces existentially quantified variables with an explicit choice function that computes the variables, based on all the arguments it depends on.

Definition 3.3.2 (Skolemization)**.** *Any prenex formula ϕ can be converted into an* equally satisfiable *skolem formula $\forall x_1 \ldots \forall x_n \psi$. This is done by repeatedly applying the following transformation, while existential quantifiers still remain:*

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

Note that f of arity n is a new *function symbol that computes y.*

A skolemized formula can be directly transformed into clausal normal form.

[3]Note that, from now on, we will write $\mathfrak{M} \models \phi$ instead of $\mathfrak{M} \models_I \phi$.

Definition 3.3.3 (Clausal Normal Form (*CNF*)). *Any skolem formula* $F \equiv \forall x_1 \ldots \forall x_n \phi$ *can be transformed into a clausal form* $CNF(F) = C$, *where* $C \equiv \bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} L_{ij}$, *i.e, as a conjunction of clauses, where* L_{ij} *are literals. This is done by:*

- *converting to equivalent formulas, until no further simplifications are possible*
- *dropping the quantifier prefix of the obtained formula.*

Horn clauses, presented next, are a subclass of clauses particularly well-tailored for formalization purposes. As we will see, this is due to the fact that that Horn satisfiability can be easily established via resolution alhorithms.

Definition 3.3.4 (Horn formulas/clauses). *Let* $F \in F_\Sigma(\mathcal{X})$ *and* $C \equiv \bigwedge_{i=1}^{m} \bigvee_{j=1}^{k_i} L_{ij}$ *its clausal normal form, i.e,* $CNF(F) = C$. *If, for all* $i \in [1, m]$, *there exists at most one* $j \in [1, k_i]$, *such that* L_{ij} *is positive,* F *is said to be a* Horn formula. *Similarly, a clause* $C = \bigvee_{j=1}^{k_i} L_{ij}$, *with at most one positive literal, is called a* Horn clause.

These types of clauses will be revisited in Section 5.

3.4. Inference

The *judgments* of a given theory can be derived via *inference systems*, as defined below.

Definition 3.4.1 (Inference System). *An* inference system $\mathcal{I}$ *for a language* $\mathcal{L}$ *consists of a set of* judgements *and (labeled)* inference rules *of the form*

$$\frac{J_1 \quad \ldots \quad J_n}{J_{n+1}} \; R$$

stating that J_{n+1} *is the* syntactical consequence *of the hypothesis* $J_1, \ldots, J_n$. *This is denoted as* $J_1, \ldots, J_n \vdash_R J_{n+1}$. *If* $n = 0$, *the judgments are called* axioms. *The iterative application of inference rules is called a* derivation.

Example 3.4.2 (Natural Deduction Inference). *The natural deduction inference system for first-order logic, consists of rules that, for the most part,* introduce *and* eliminate *operators, as illustrated below for conjunction and implication.*

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B} \; \wedge \text{ intro} \qquad \frac{\vdash A \wedge B}{\vdash A} \; \wedge \text{ elim left} \qquad \frac{\vdash A \wedge B}{\vdash B} \; \wedge \text{ elim right}$$

$$\frac{A \vdash B}{A \Rightarrow B} \; \Rightarrow \text{ intro} \qquad \frac{A \Rightarrow B \quad A}{B} \; \Rightarrow \text{ elim (modus ponens)}$$

The following properties link inference systems to the semantics of the language they are defined over, by stating that all syntactic consequences are semantically valid (soundness) and vice-versa (completeness).

Definition 3.4.3 (Soundness and Completeness)**.** *Let $\mathcal{I}$ be an inference system for a language $\mathcal{L}$. If, for all $\Delta \subset SEN_{\mathcal{L}}, F \in SEN_{\mathcal{L}}$:*

$\Delta \vdash_{\mathcal{I}} F$ *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 the context of studying formal languages, in addition to natural deduction, other inference systems have been developed, such as Hilbert-style systems, sequent systems, production grammars and type theories. However, the rise of computers and mechanized theorem proving created the necessity of *searching* for derivations via *efficient* proof methods. As a result, *resolution* based inference techniques were introduced. These operate on *clausal formulas* and, due to their epurated form, lend themselves particularly well to *mechanization*. We illustrate below one of the simplest such systems, namely *binary resolution*.

Example 3.4.4 (Binary Resolution Inference)**.** *The system consists of two rules :*

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

where σ is the most general unifier (mgu) of C and D ($\sigma = mgu(C, D)$), i.e, $\sigma C = \sigma D$

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

where $\sigma = mgu(B, C)$, i.e, $\sigma B = \sigma C$.

Note: binary resolution *is a generalization of the* modus ponens *from Example 3.4.2*

While many versions of resolution exist in the literature (see [7]), we focus on *hyperresolution* (due to [75]), revisited in Section 5.

Definition 3.4.5 (Hyperresolution Inference)**.** *The* hyperresolution *rule is given by:*

$$\frac{C_1 \vee A_1 \ldots C_n \vee A_n \quad D \vee \neg B_1 \vee \ldots \vee \neg B_n}{\sigma(C_1 \vee \ldots \vee C_n \vee D)}$$

where, for all $i \in [1, n]$, $\sigma = mgu(A_i, B_i)$, i.e, $\sigma A_i = \sigma B_i$.

Note that hyperresolution can be seen as *iterated binary resolution*. Also, it can be made more efficient, by adding additional *heuristics*, regarding ordering and selection.

Theorem 3.4.6. *Hyperresolution is* sound *and* refutationally complete.

Proof. See [7]. □

4. The Relational Model

Databases have been developed as viable solutions for accurate and efficient *storage* and *retrieval* of large amounts of data. Formally representing organized collections of related data, the notion of a *database* is intertwined, at an abstract level, with that of a *data model*. As such, it is the underlying data model of a database that provides the necessary formalism for defining, modifying and accessing its stored information. Historically, the first data models were flat file systems that relied on sequential files linearly storing fixed-length records. Among the many drawbacks of this approach, the more prominent were *domain-specificity*, *redundancy*, *dependency* on storage devices, *high maintenance* and *weak security*. The emergence of *database systems* was directly aimed at overcoming such limitations. These systems relied on the *relational model* introduced in [23]. First implemented in systems such as INGRES and System/R, it is still the basis for most current DBMSs, such as Oracle, Microsoft SQL Server, IBM DB2, Sybase, PostgreSQL, MySQL. In this chapter, we present the fundamentals of the relational model, as a tool for *representing* and *extracting* data (see Section 4.1 and Section 4.2), while enforcing its *integrity* (see Section 4.3). Our presentation is based on that given in [1].

4.1. Data Representation

The relational model centers on representing data structures as mathematical *relations*.

Definition 4.1.1 (Domains)**.** *A* domain $\mathcal{D}$ *is a set of constant values.*

Definition 4.1.2 (Cartesian Product)**.** *The n-ary* Cartesian product *over domains* $\mathcal{D}_1, \ldots, \mathcal{D}_n$ *is the set of all ordered n-tuples* $(v_1, \ldots, v_n)$*, such that each tuple component ranges over its respective domain, i.e* $v_i \in \mathcal{D}_i$*, for all* $i \in [1, n]$*.*

$$\mathcal{D}_1 \times \ldots \times \mathcal{D}_n \equiv \{(v_1, \ldots, v_n) \mid v_1 \in \mathcal{D}_1, \ldots, v_n \in \mathcal{D}_n\}$$

In the relational model, data is intuitively represented by *tables* (tabulated relations) consisting of rows (tuples), with uniform *structure* and intended meaning, each of which gives information about a specific entity. In practice, columns (tuple components) are labeled with names called *attributes*. An important distinction is made between the *structure* (schema) and the *content* (instance) of such tables.

Assume finite sets of *attributes* att and relation names $\mathsf{relname}$, related by the sorting function $\mathsf{sort} : \mathsf{relname} \to 2^{\mathsf{att}}$.

Title	Director	Actor
An American in Paris	Minnelli	Kelly
An American in Paris	Minnelli	Caron
Blue Velvet	Lynch	Hopper
Blue Velvet	Lynch	MacLachlan
Blue Velvet	Lynch	Rossellini
M	Lang	Lorre
Senso	Visconti	Granger

Table 4.1.: **Movies** relation

Theater	Title	Schedule
Action Christine	Senso	20
Action Christine	Blue Velvet	14
Le Champo	M	16
Action Christine	An American in Paris	16

Table 4.2.: **Location** relation

Theater	Address	Phone
Action Christine	4 rue Christine, Paris 6	0143258578
Le Champo	51 rue des Ecoles, Paris 5	0143545160

Table 4.3.: **Pariscope** relation

Definition 4.1.3 (Database Schemas)**.** *A* relation schema $R[U]$ *consists of a relation name* $R \in \mathsf{relname}$ *and of set* $U \subseteq \mathsf{att}$*, such that* $\mathsf{sort}(R) = U$*. A* database schema $\mathbf{R}$ *is a non-empty finite set of* relation schemas*,* $\mathbf{R} = \{R_1[U_1], \ldots, R_n[U_n]\}$*, with* $\mathsf{sort}(R_i) = U_i$*.*

Definition 4.1.4 (Database Instances)**.** *A* relation instance I *of a schema* $R[U]$ *over a domain* $\mathcal{D}$ *is a finite set of (stored) tuples* $I \subseteq \mathcal{D}^{|U|}$*. A* database instance $\mathbf{I}$ *of a schema* $\mathbf{R}$*, over a domain* $\mathcal{D}$*, is a family of relation instances, i.e,* $\mathbf{I} = \{I_i\}_{1 \leq i \leq |\mathbf{R}|}$*.*

To illustrate, let us consider the **Movies** relation above, describing the movies on display at a given cinema, during a certain week. The *relation schema* is denoted as **Movies [Title, Director, Actor]**, where $\mathsf{sort}(\mathbf{Movies}) = \{\mathbf{Title}, \mathbf{Director}, \mathbf{Actor}\}$. As such, each table row in the *relation instance* stores a movie's title, director and one of its lead actors. Note that, for a given movie, there are as many entries as lead actors.

A relational database is represented by a collection of tables. As defined previously, a *database schema* is given by a nonempty countable set of sorted relation names, while a *database instance* refers to the collection of corresponding relation instances. Extrapolating from the previous example, we can imagine a **Cinema** database that stores the weekly movie program of each cinema in Paris, as listed in the Pariscope journal[1]. The

[1]The Pariscope is a Parisian journal for advertising cultural events.

locations could be characterized by a theater name, an address and a phone number and the journal program could contain the theater name, movie title and schedule.

$$\begin{aligned} \mathbf{Cinema} &= \{\mathbf{Movies}, \mathbf{Location}, \mathbf{Pariscope}\} \\ \\ \mathsf{sort}(\mathbf{Movies}) &= \{\mathbf{Title}, \mathbf{Director}, \mathbf{Actor}\} \\ \mathsf{sort}(\mathbf{Location}) &= \{\mathbf{Theater}, \mathbf{Address}, \mathbf{Phone}\} \\ \mathsf{sort}(\mathbf{Pariscope}) &= \{\mathbf{Theater}, \mathbf{Title}, \mathbf{Schedule}\} \end{aligned}$$

Figure 4.1.: **Cinema** database schema

4.2. Data Extraction

Information is extracted from databases through mechanisms called *queries*. These are expressed through *query languages*, a variety of which are supported by the relational model. At their core, these are fundamentally rooted in two classes of *equivalent* mathematical formalisms that stem from *algebraic* and, respectively, *logical* approaches. The first is illustrated by the *relational algebra*, which is more *operational* and describes *how* needed information can be computed. The second is represented by the *relational calculus*, which is more *declarative* and describes *what* is the nature of the required information. We give an overview of these languages in Section 4.2.1 and Section 4.2.2.

4.2.1. Relational Algebra

Relational algebra is an umbrella term for a family of algebraic query languages. These are defined over relation instances and extend the algebra of sets with specific operators. Depending on whether these operators refer to attributes *positionally* or *nominally*, relational algebras are said to either follow an *unnamed* or a *named* perspective. For completeness purposes, we overview both, as exemplified by the SPC and SPJR algebras. Note however that the corresponding Coq formalization in Section 7.1 favours the latter approach, as it is more prevalent in practice.

The SPC Algebra

The primitive operators of the SPC algebra are selection (σ), projection (π) and the cross-product ($\times$). We formally define them below, given relation instances I and J of arity n and m over a domain $\mathcal{D}$.

$$\begin{array}{lll} \mathsf{Selection} & \sigma_{j=a}(I) & = \{t \in I \mid t(j) = a\},\ j \leq n,\ a \in \mathcal{D} \\ \mathsf{Projection} & \pi_{j_1,\ldots,j_n}(I) & = \{(t(j_1), \ldots, t(j_n)) \mid t \in I\} \\ \mathsf{Cross\text{-}product} & I \times J & = \{(t_1(1), \ldots, t_1(n), t_2(1), \ldots, t_2(m)) \mid t_1 \in I,\ t_2 \in J\} \end{array}$$

The SPJR Algebra

The primitive operators of the SPJR algebra are selection (σ), projection (π), join ($\bowtie$) and renaming (ρ). We formally define them below, given a global set of attributes **att**, a domain $\mathcal{D}$ and relation instances I, J of sorts $V, W \subseteq \mathbf{att}$. Note that, for a tuple t and an attribute set A, we use $t|_A$ to represent the tuple obtained from t by keeping only attributes in A.

Selection	$\sigma_{j=a}(I)$	$=$	$\{t \mid t \in I \wedge t(j) = a\},\ j \in V, a \in \mathcal{D} \cup V$
	$\sigma_{i=j}(I)$	$=$	$\{t \in I \mid t(i) = t(j)\},\ i, j \in V$
Projection	$\pi_A(I)$	$=$	$\{t|_A \mid t \in I\},\ A \subseteq V$
Join	$I \bowtie J$	$=$	$\{t \mid \exists v \in I, \exists w \in J, t|_V = v \text{ and } t|_W = w\}$
Rename	$\rho_g(I)$	$=$	$\{t \mid \exists u \in I, \forall a \in V, t(g(a)) = u(a)\},\ g : V \to \mathbf{att}$

4.2.2. Relational Calculus

Relational calculus is a logical query language centered on the insight that database relations can be viewed as *interpretations of predicates* from first-order logic. Consequentely, it is a specialization of the first-order languages reviewed in Section 3. Unlike these, however, the relational calculus is concerned only with *validity* in a fixed model, given by the current database state. Moreover, in order to exclude infinite interpretations, relational calculus formulas are required to be *domain-independent* (see Definition 4.2.3). Since this property is *undecidable* (see [32]), syntactical restrictions are imposed, called *safety conditions*. The relational calculus has two declinations, *tuple relational calculus* and *domain relational calculus*, that differ on whether variables range over tuples or domain values. In this section, we focus on *conjunctive queries*, a subclass of the domain relational calculus, for which efficient optimization techniques exist.

Let $\mathcal{L}$ be a first-order language over a domain $\mathcal{D}$, whose signature contains arity predicates that are either relation names or comparison predicates. The syntax of the relational calculus language extends $\mathcal{L}$ with *query* expressions, as defined below.

Definition 4.2.1 ((Safe) Relational Calculus Queries)**.** *Let $F \in F_\Sigma(\mathcal{X})$ be a formula and $\vec{t} = (u_1, \ldots, u_n)$, a tuple over a schema $\mathbf{R}$, such that $\vec{t} \in T_\Sigma(\mathcal{X})^n$. A* query *$Q$ is an expression $Q = \{\vec{t} \mid F\}$. If Q satisfies the* safety condition $\bigcup_{i=1}^{n} FV(u_i) \subseteq FV(F)$*, then Q is said to be* safe*.*

Databases can be seen as Herbrand Interpretations (see Definition 3.2.1) over $\mathcal{L}$. Let F be a given $\mathcal{L}$-formula and $\mathbf{I}$, a database instance over a schema $\mathbf{R}$. The interpretation $[\![F]\!]^{\mathbf{I}}$ of F with respect to $\mathbf{I}$ is analogous to that described in Section 3. Hence, we can define the semantics of a query $Q = \{\vec{t} \mid F\}$ over $\mathbf{R}$, as establishing whether $\mathbf{I} \models F$.

The main problems concerning (relational calculus) queries are computing their evaluation and establishing, given two queries, whether one is contained in the other and

whether they are equivalent. We present the corresponding notions of evaluation, containment and equivalence in Definitions 4.2.2, 4.2.5 and 4.2.6.

Definition 4.2.2 (Relational Calculus Query Evaluation)**.** *Let* $\mathbf{I}$ *be a database instance over a schema* $\mathbf{R}$ *and domain* $\mathcal{D}$*. Let* $Q \equiv \{(u_1, \ldots, u_n) \mid F\}$ *be a relational calculus query on* $\mathbf{I}$*. The* query evaluation $Q(\mathbf{I}, \mathcal{D})$*, also denoted as* $[\![Q]\!]^{\mathbf{I}}$*, is:*

$$Q(\mathbf{I}, \mathcal{D}) = \{(\nu(u_1), \ldots, \nu(u_n)) \mid \nu : \bigcup_{i=1}^{n} FV(u_i) \to \mathcal{D} \wedge [\![F]\!]^{\mathbf{I},\nu} = \top\}.$$

Definition 4.2.3 (Domain-Independence)**.** *Let* $\mathbf{I}$ *be a database instance over a schema* $\mathbf{R}$*. A relational calculus query* Q *is said to be* domain independent*, if, for any distinct arbitrary domains,* $\mathcal{D}_1$ *and* $\mathcal{D}_2$*:*

$$Q(\mathbf{I}, \mathcal{D}_1) = Q(\mathbf{I}, \mathcal{D}_2)$$

Remark 4.2.4. *Safe Relational Calculus is domain independent.*

Definition 4.2.5 (Relational Calculus Query Containment)**.** *Let* $\mathbf{R}$ *be a database schema. Let* Q_1 *and* Q_2 *be two relational calculus queries over* $\mathbf{R}$*.* Q_1 *is contained in* Q_2*, denoted as* $Q_1 \subseteq Q_2$*, if, for all instances* $\mathbf{I}$ *of domain* $\mathcal{D}$*:*

$$Q_1(\mathbf{I}, \mathcal{D}) \subseteq Q_2(\mathbf{I}, \mathcal{D})$$

Query inclusion naturally induces an equivalence, as follows.

Definition 4.2.6 (Relational Calculus Query Equivalence)**.** *Let* $\mathbf{R}$ *be a database schema. Let* Q_1 *and* Q_2 *be two relational calculus queries over* $\mathbf{R}$*.* Q_1 *is equivalent to* Q_2*, denoted as* $Q_1 \equiv Q_2$*, if, for all instances* $\mathbf{I}$ *of domain* $\mathcal{D}$*:*

$$Q_1(\mathbf{I}, \mathcal{D}) \subseteq Q_2(\mathbf{I}, \mathcal{D}) \text{ and } Q_2(\mathbf{I}, \mathcal{D}) \subseteq Q_2(\mathbf{I}, \mathcal{D})$$

The containment and equivalence problems for relational calculus queries (and for their relational algebra counterparts) are undecidable ([77]). Hence, restricted fragments of the full relational calculus, for which these properties can be decided, have been studied. The conjunctive queries, overviewed next, are one such subclass.

Conjunctive Queries

Conjunctive queries, first introduced in [20], are a subclass of the *domain relational calculus*. Specifically, they describe the query language for which query containment and equivalence are inherently decidable. Conjunctive queries are expressed in relational calculus form, as in Definition 4.2.7. They are also translatable into SPJ relational algebra and into tableau form, as shown in Examples 4.2.9 and 4.2.11.

Definition 4.2.7 ((Safe) Conjunctive Queries)**.** *A* conjunctive query Q *over a database schema* $\mathbf{R} = \{R_1, \ldots, R_k\}$ *has the form* $Q = \{\vec{t} \mid \exists \vec{X},\ R_1(\vec{t_1}) \wedge \ldots \wedge R_k(\vec{t_k})\}$*. If* Q *satisfies the* safety condition $\vec{X} \subseteq (\bigcup_{i=1}^{n} FV(\vec{t_i}) \setminus FV(\vec{t}))$*, then* Q *is said to be* safe*.*

In rule notation, the query corresponds to $Q(\vec{t}) :- R_1(\vec{t_1}), \ldots, R_k(\vec{t_k})$ (see Chapter 5).

Example 4.2.8 (Conjunctive Query over the **Cinema** database)**.** *The query:*

'Which of Fellini's movies are played at the cinema "Action Christine" ?"

over the database schema in Figure 4.1, corresponds to the relational calculus *expression:*

$$\left\{ (t,d,a) \mid \begin{array}{l} \exists th, \exists t', \exists s, \mathbf{Movies}(t,d,a) \wedge \mathbf{Pariscope}(th,t',s) \\ \wedge\, t = t' \wedge d = \textit{"Fellini"} \wedge th = \textit{"Action Christine"} \end{array} \right\}$$

Example 4.2.9 (Translation of a Conjunctive Query into a SPJ Algebra Expression)**.** *The query above can be translated into the* SPJ relational algebra *as:*

$$\pi_{\{\textit{Title, Director, Actor}\}}(\sigma_{\substack{\textit{x.Director = "Fellini"} \wedge \\ \textit{x.Theater = "Action Christine"}}} (\mathbf{Movies} \bowtie \mathbf{Pariscope}))$$

Definition 4.2.10 (Tableau Representation)**.** *According to [1], a* tableau T *over a schema* $\mathbf{R}$ *is an instance that contains* extended tuples*, i.e tuples mapping schema attributes to either constants or variables. As such, conjunctive queries of the form given in Definition 4.2.7, can be represented by a pair* (T, s)*, as follows:*

- *a first row contains the relevant attributes from* $\mathsf{sort}(\mathbf{R})$
- *the* rows *in* T *contain blanks and the components of conjunction atoms* $R_i(\vec{t_i})$*, whose existentially quantified variables* $\vec{X}$ *are called* nondistinguished
- *a final row* s*, called* summary*, contains blanks and the components of* $\vec{t}$*, whose variables are called* distinguished*.*

Example 4.2.11 (Translation of a Conjunctive Query into Tableau Representation)**.** *The query in Example 4.2.8 can be translated into tableau form as:*

Title	Director	Actor	Theater	Schedule	
t	"Fellini"	a			Movies
t			"Action Christine"	s	Pariscope
t	d	a			summary

Note that the corresponding equalities have been embedded.

4.3. Data Integrity

An important aspect to be taken into account by data models, in general, and the relational model, in particular, is ensuring the *integrity* of the underlying stored data, i.e its *accuracy* and *consistency*. However, the data representation language in itself acts solely as a syntactic means of structuring and relating information, not of describing

its nature. To remedy this, further semantic specifications, called *integrity constraints*, were introduced to complement the database schema. These are logical properties that precisely describe the application-specific meaning of stored data. Thus, they restrict the set of possible database instances, by weeding out *anomalies*.

Since their study was first pioneered by Codd in [23], a plethora of different classes of integrity constraints were developed during more than a decade. Among these, the most prominent are *functional*, *join* and *inclusion* dependencies. This proliferation of data integrity formalisms culminated with the insight that first-order logic provided a general and unifying framework. As such, integrity constraints were polarized into *tuple-generating*, universally quantified formulas enforcing tuple existence, and *equality-generating*, universally quantified formulas enforcing variable equality.

Once placed on solid foundations, dependency theory research focused on the main problem of *logical implication*:

"Given a set of dependencies Σ and a dependency d, is d implied by Σ, i.e, $\Sigma \models d$?".

To answer this, two complementary approaches were proposed: one based on constructing finite axiomatizations for *proving* implication and the other, based on constructing algorithms for *testing* implication. The former method led to the development of inference systems for different classes of dependencies, the most prominent of which is *Armstrong's system*, for *functional dependencies*. The latter resulted in the development of a family of *chase* procedures, for more *general dependencies*.

This section is structured to reflect both these aspects of the logical implication problem. In Section 4.3.1, we begin from a narrower scope, formally introducing functional dependencies, their axiomatization and its properties. In Section 4.3.3, we frame this in a wider setting, by presenting general dependencies, the chase procedure and its properties.

4.3.1. Logical Implication for Functional Dependencies

Functional Dependencies

Let $R[U]$ be a relation schema. A *functional dependency (FD)* over $R[U]$ is an expression that relates attribute sets $V, W \subseteq U$, denoted as $V \hookrightarrow W$. Informally, $V \hookrightarrow W$ *holds* on $R[U]$ if, for any instance I of R, any two tuples in I having equal components on V[2], also have equal components on W.

Definition 4.3.1 (FD Satisfiability)**.** *A relation instance I over $R[U]$ is said to* satisfy *$V \hookrightarrow W$, denoted as $I \models V \hookrightarrow W$, if*

$$\forall t_1 \forall t_2, t_1 \in I \Rightarrow t_2 \in I \Rightarrow t_1|_V = t_2|_V \Rightarrow t_1|_W = t_2|_W$$

Let F be a set of functional dependencies. I is said to satisfy *F, denoted $I \models F$, if*

[2]meaning that the tuples have the same component values for all attributes in V.

$$\forall d, d \in F \Rightarrow I \models d.$$

Let us fix F and $\tilde{F}$, sets of functional dependencies over U.

Definition 4.3.2 (FD Implication)**.** *$\tilde{F}$ is* logically implied *by F, denoted $F \models \tilde{F}$, if*

$$\forall I : U, I \models F \Rightarrow I \models \tilde{F}.$$

Definition 4.3.3 (FD Closure)**.** *The set F^+ of all functional dependencies that are logically implied by F is called its* closure *and is defined as $F^+ = \{d \mid F \models d\}$.*

Definition 4.3.4 (FD Equivalence)**.** *$\tilde{F}$ and F are* logically equivalent*, denoted $F \equiv \tilde{F}$, if $F \models \tilde{F}$ and $\tilde{F} \models F$. Note that $F \equiv \tilde{F} \Leftrightarrow F^+ = \tilde{F}^+$.*

As can be seen from the previous definitions, closures are important in establishing whether implication or equivalence hold between arbitrary sets of functional dependencies. Indeed, given a procedure for constructing closures, checking these properties translates to respectively testing set membership and equality. To this end, particularly relevant is *Armstrong's System*, a *sound* and *complete* axiomatization presented in [5], as a way of characterizing functional dependency implication.

Armstrong's System

The set $\mathcal{A}$ of inference rules proposed by Armstrong is given by the rules in Figure 4.2.

$$\frac{Y \subseteq X}{X \hookrightarrow Y}\ \mathsf{FD}_1 : \text{reflexivity} \qquad \frac{X \hookrightarrow Y}{X \cup Z \hookrightarrow Y \cup Z}\ \mathsf{FD}_2 : \text{augmentation}$$

$$\frac{X \hookrightarrow Y \quad Y \hookrightarrow Z}{X \hookrightarrow Z}\ \mathsf{FD}_3 : \text{transitivity}$$

Figure 4.2.: Armstrong's System

Definition 4.3.5 (FD Inference)**.** *Let F be a set of functional dependencies over a given schema and $\mathcal{A}$ be Armstrong's system of rules. A functional dependency $X \hookrightarrow Y$ can be inferred from F using $\mathcal{A}$, denoted $F \vdash_{\mathcal{A}} X \hookrightarrow Y$, if there is a proof derivation for $X \hookrightarrow Y$ from $\mathcal{A}$, using the dependencies in F as axioms.*

We can test if $F \vdash_{\mathcal{A}} X \hookrightarrow Y$, based on the Armstrong closure of X, defined below.

Definition 4.3.6 (Armstrong Closure)**.** *Let U be an attribute set and $X \subseteq U$.*
The closure X_F^+ of X with respect to a set of functional dependencies F over U is

$$X_F^+ = \bigcup \{W \subseteq U \mid F \vdash_{\mathcal{A}} X \hookrightarrow W\}.$$

Hence, $F \vdash_{\mathcal{A}} X \hookrightarrow Y$ iff $Y \subseteq X_F^+$. The Armstrong closure can be computed as follows.

```
1: procedure CLOSURE(X, F)
2:     X_0 ← X
3:     i ← 0
4:     while X_i ≠ X_{i+1} do
5:         for all V ↪ W ∈ F do
6:             if V ⊆ X_i and W ⊈ X_i then
7:                 X_i ← X_i ∪ W
8:                 i ← i + 1
9:     return X_i
```

Theorem 4.3.7 (Armstrong Soundness)**.** *Given a set of dependencies F over an attribute set U and attribute sets $X, Y \subseteq U$, if $F \vdash_{\mathcal{A}} X \hookrightarrow Y$, then $F \models X \hookrightarrow Y$.*

Proof. The proof is by induction on the length of the derivation. In the base case, we prove soundness of each inference rule. To this end, let us fix I, a relation instance over U, and arbitrary tuples $t_1, t_2 \in I$.

Reflexivity Assume $Y \subseteq X$. If $t_1|_X = t_2|_X$, trivially $t_1|_Y = t_2|_Y$. Hence, $I \models X \hookrightarrow Y$.

Augmentation Assume $I \models X \hookrightarrow Y$. If $t_1|_{X \cup Z} = t_2|_{X \cup Z}$, then $t_1|_X = t_2|_X$. This implies $t_1|_Y = t_2|_Y$ and, by extension, $t_1|_{Y \cup Z} = t_2|_{Y \cup Z}$. Hence, $I \models X \cup Z \hookrightarrow Y \cup Z$.

Transitivity Assume $I \models X \hookrightarrow Y$ and $I \models Y \hookrightarrow Z$. If $t_1|_X = t_2|_X$, from $I \models X \hookrightarrow Y$, it follows that $t_1|_Y = t_2|_Y$ and, by $I \models Y \hookrightarrow Z$, that $t_1|_Z = t_2|_Z$. Hence, $I \models X \hookrightarrow Z$.

□

From the soundness of Armstrong's System, other rules, like the ones in Figure 4.3, can also be proven. These allow to establish a key result, relating the Armstrong closure of attribute sets with functional dependency inference, as stated in Lemma 4.3.8.

$$\frac{X \hookrightarrow Y \quad X \hookrightarrow Z}{X \hookrightarrow Y \cup Z}\ \mathsf{FD_4} : \mathsf{union} \qquad \frac{X \hookrightarrow Y \quad Z \subseteq Y}{X \hookrightarrow Z}\ \mathsf{FD_5} : \mathsf{decomposition}$$

Figure 4.3.: Admissible Rules for Functional Dependency Inference

Lemma 4.3.8. *Given the attribute set U and $X, Y \subseteq U$, $F \vdash_{\mathcal{A}} X \hookrightarrow Y \Leftrightarrow Y \subseteq X^+$.*

Proof. Let $Y = \{A_1, \ldots, A_n\}$.

$\Rightarrow$ Definition 4.3.6 implies $F \vdash_{\mathcal{A}} X \hookrightarrow A_i$, for all i. Hence, by the union rule, $X \hookrightarrow Y$.

$\Leftarrow$ From $X \hookrightarrow Y$, by the decomposition rule, $F \vdash_{\mathcal{A}} X \hookrightarrow A_i$, for all i. Hence, $Y \subseteq X^+$.

□

This lemma is crucial in the following completeness proof for Armstrong's system.

Theorem 4.3.9 (Armstrong Completeness)**.** *Given a set of dependencies F over an attribute set U and attribute sets $X, Y \subseteq U$, if $F \models X \hookrightarrow Y$, then $F \vdash_{\mathcal{A}} X \hookrightarrow Y$.*

Proof. We prove the contrapositive statement: if $F \nvdash_{\mathcal{A}} X \hookrightarrow Y$ then $F \not\models_{\mathcal{A}} X \hookrightarrow Y$, i.e $\exists I$, such that $I \models F$ and $I \not\models X \hookrightarrow Y$. Let I consist of tuples t_1 and t_2, as shown below.

$$\begin{array}{cccc} & \overbrace{}^{X^+} & \\ t_1 & 1\,1\,1\,1 & 0\ldots0 \\ t_2 & 1\,1\,1\,1 & 1\ldots1 \end{array}$$

First, we prove $I \models F$, i.e $\forall V \hookrightarrow W,\ V \hookrightarrow W \in F \Rightarrow I \models V \hookrightarrow W$. Assuming $t_1|_V = t_2|_V$, we show $t_1|_W = t_2|_W$. From the definition of I, $t_1|_V = t_2|_V \Rightarrow V \subseteq X^+$. According to Lemma 4.3.8, $F \vdash_{\mathcal{A}} X \hookrightarrow V$, which, together with $F \vdash_{\mathcal{A}} V \hookrightarrow W$, via transitivity, produce $F \vdash_{\mathcal{A}} X \hookrightarrow W$ (4.1). From reflexivity, it is inferred that $\forall A, A \in W \Rightarrow \ F \vdash_{\mathcal{A}} W \hookrightarrow A$ (4.2). From (4.1) and (4.2), via transitivity, we obtain $F \vdash_{\mathcal{A}} X \hookrightarrow A$ (4.3). Since $A \in X^+$, for all $A \in W$, $W \subseteq X^+$. Hence, $t_1|_W = t_2|_W$.

Next, we prove $I \not\models X \hookrightarrow Y$. As $X \subseteq X^+$ and $F \nvdash_{\mathcal{A}} X \hookrightarrow Y$, it follows that $Y \nsubseteq X^+$. □

4.3.2. Logical Implication for Multivalued Dependencies

Multivalued Dependencies

Let $R[U]$ be a relation schema. A *multivalued dependency (MD)* over $R[U]$ is an expression that relates attribute sets $V, W \subseteq U$, denoted as $V \twoheadrightarrow W$. Informally, $V \twoheadrightarrow W$ *holds* on $R[U]$ if, for any instance I of $R[U]$, we can swap the components in W of any two tuples whose components agree on V, to obtain two tuples also in I.

Definition 4.3.10 (MD Satisfiability)**.** *A relation instance I over $R[U]$ is said to* satisfy *$V \twoheadrightarrow W$, denoted as $I \models V \twoheadrightarrow W$, if*

$$\forall t_1 \forall t_2, t_1 \in I \Rightarrow t_2 \in I \Rightarrow t_1|_V = t_2|_V \Rightarrow \exists t_3, t_3 \in I \wedge t_3|_V = t_1|_V \wedge t_3|_W = t_1|_W \wedge t_3|_Z = t_2|_Z$$

where $Z = U \setminus (V \cup W)$.

Note that the existence of the second tuple $t_4 \in I$, such that

$$t_4|_V = t_2|_V \wedge t_4|_W = t_2|_W \wedge t_4|_Z = t_1|_Z,$$

follows from interchanging t_1 and t_2.

Inference System for Multivalued Dependencies

Analogous to what was presented in Section 4.3.2, in order to determine whether a given multivalued dependency logically implies another, one can appeal to the inference system presented in Figure 4.4.

$$\frac{X \twoheadrightarrow Y}{X \twoheadrightarrow U \setminus (X \cup Y)}\ \mathsf{MVD_0}\text{ : complementation} \qquad \frac{Y \subseteq X \subseteq U}{X \twoheadrightarrow Y}\ \mathsf{MVD_1}\text{ : reflexivity}$$

$$\frac{X \twoheadrightarrow Y \quad Z \subseteq U}{X \cup Z \twoheadrightarrow Y \cup Z}\ \mathsf{MVD_2}\text{ : augmentation} \qquad \frac{X \twoheadrightarrow Y \quad Y \twoheadrightarrow Z}{X \twoheadrightarrow Z}\ \mathsf{MVD_3}\text{ : transitivity}$$

Figure 4.4.: Inference System for Multivalued Dependencies

The following admissible rules relate multivalued and functional dependencies:

$$\frac{X \hookrightarrow Y}{X \twoheadrightarrow Y}\ \mathsf{FMVD_1}\text{ : conversion} \qquad \frac{X \twoheadrightarrow Y \quad X \cup Y \hookrightarrow Z}{X \hookrightarrow Z \setminus Y}\ \mathsf{FMVD_2}\text{ : interaction}$$

Figure 4.5.: Admissible Rules for Multivalued Dependency Inference

Theorem 4.3.11. *The inference system consisting of the rules in Figures 4.2, 4.4 and 4.5 is* sound *for the logical implication of functional and multivalued dependencies considered together.*

Proof. The proof is by induction on the length of derivation. In the base case, we prove soundness of each inference rule. To this end, let us fix I, a relation instance over U and tuples $t_1, t_2 \in I$.

Complementation Assume $I \models X \twoheadrightarrow Y$. If $t_1|_X = t_2|_X$, then there exists a tuple $t_3 \in I$, such that $t_3|_X = t_1|_X$, $t_3|_Y = t_1|_Y$ and $t_3|_{U\setminus(X\cup Y)} = t_2|_{U\setminus(X\cup Y)}$. Since $t_3|_X = t_2|_X$, $t_3|_Y = t_2|_{U\setminus(X\cup Y)}$ and $t_3|_Y = t_1|_Y$, it follows that $I \models X \twoheadrightarrow U \setminus (X \cup Y)$.

Reflexivity Assume $Y \subseteq X$. Then, if $t_1|_Y = t_2|_Y$, it holds that $t_1|_X = t_2|_X$. To establish $I \models X \twoheadrightarrow Y$, we have to provide a witness tuple $t_3 \in I$, such that $t_3|_X = t_1|_X$, $t_3|_Y = t_1|_Y$ and $t_3|_{U\setminus(X\cup Y)} = t_2|_{U\setminus(X\cup Y)}$. Taking t_3 to be t_2 concludes the proof.

Augmentation Assume $I \models X \twoheadrightarrow Y$. If $t_1|_X = t_2|_X$, then there exists a tuple $t_3 \in I$, such that $t_3|_X = t_1|_X$, $t_3|_Y = t_1|_Y$ and $t_3|_{U\setminus(X\cup Y)} = t_2|_{U\setminus(X\cup Y)}$. It follows that $t_3|_{X\cup Z} = t_1|_{X\cup Z}$, $t_3|_{Y\cup Z} = t_1|_{Y\cup Z}$ and, respectively, $t_3|_{U\setminus(X\cup Y\cup Z)} = t_2|_{U\setminus(X\cup Y\cup Z)}$. Hence, $I \models X \cup Z \twoheadrightarrow Y \cup Z$.

Transitivity Assume $I \models X \twoheadrightarrow Y$ and $I \models Y \twoheadrightarrow Z$. Let $V = U \setminus (X \cup Y \cup Z)$.
If $t_1|_X = t_2|_X$, from $I \models X \twoheadrightarrow Y$, there exists a tuple $t_3 \in I$, such that $t_3|_X = t_1|_X$, $t_3|_Y = t_1|_Y$ and $t_3|_{U\setminus(X\cup Y)} = t_2|_{U\setminus(X\cup Y)}$, i.e, $t_3|_{V\cup Z} = t_2|_{V\cup Z}$. From $t_3|_Y = t_1|_Y$ and $I \models Y \twoheadrightarrow Z$, it holds that there exists a tuple $t_4 \in I$, such that $t_4|_Y = t_3|_Y$, $t_4|_Z = t_3|_Z$ and $t_4|_{U\setminus(Y\cup Z)} = t_1|_{U\setminus(Y\cup Z)}$, i.e, $t_4|_{V\cup X} = t_1|_{V\cup X}$. From the latter, we have $t_4|_X = t_1|_X$, which, together with $t_1|_X = t_2|_X$, leads to $t_4|_X = t_2|_X$. As $t_3|_{V\cup Z} = t_2|_{V\cup Z}$, we have $t_3|_Z = t_2|_Z$, which, together with $t_4|_Z = t_3|_Z$, gives $t_4|_Z = t_2|_Z$. Finally, from $t_4|_Y = t_3|_Y$ and $t_3|_Y = t_1|_Y$, $t_4|_Y = t_1|_Y$ and, hence, $t_3|_{V\cup Y} = t_1|_{V\cup Y}$. As $t_4|_X = t_2|_X$, $t_4|_Z = t_2|_Z$ and $t_3|_{V\cup Y} = t_1|_{V\cup Y}$, $I \models X \twoheadrightarrow Z$.

Conversion Assume $I \models X \hookrightarrow Y$. If $t_1|_X = t_2|_X$, then $t_1|_Y = t_2|_Y$. To establish $I \models X \twoheadrightarrow Y$, we have to provide a witness tuple $t_3 \in I$, such that $t_3|_X = t_1|_X$, $t_3|_Y = t_1|_Y$ and $t_3|_{U\setminus(X\cup Y)} = t_2|_{U\setminus(X\cup Y)}$. Taking t_3 to be t_2 concludes the proof.

Interaction Assume $I \models X \twoheadrightarrow Y$ and $I \models X \cup Y \hookrightarrow Z$. Let $V = U \setminus (X \cup Y \cup Z)$. If $t_1|_X = t_2|_X$, then there exists a tuple $t_3 \in I$, such that $t_3|_X = t_1|_X$, $t_3|_Y = t_1|_Y$ and $t_3|_{U\setminus(X\cup Y)} = t_2|_{U\setminus(X\cup Y)}$, i.e $t_3|_{V\cup Z} = t_2|_{V\cup Z}$. From the latter it holds that $t_3|_{Z\setminus Y} = t_2|_{Z\setminus Y}$. From $t_3|_X = t_1|_X$ and $t_3|_Y = t_1|_Y$, we have $t_3|_{X\cup Y} = t_1|_{X\cup Y}$. As $I \models X \cup Y \hookrightarrow Z$, we obtain $t_3|_Z = t_1|_Z$. Consequentely, $t_3|_{Z\setminus Y} = t_1|_{Z\setminus Y}$. Together with $t_3|_{Z\setminus Y} = t_2|_{Z\setminus Y}$. it follows $t_1|_{Z\setminus Y} = t_2|_{Z\setminus Y}$. Hence, $I \models X \hookrightarrow Z \setminus Y$.

□

Theorem 4.3.12. *The inference system consisting of the rules in Figures 4.2, 4.4 and 4.5 is* complete *for the logical implication of functional and multivalued dependencies considered together.*

Proof. See [1]. We omit the completeness proof, as it is not part of the Coq formalization we present in Chapter 7. □

4.3.3. Logical Implication for General Dependencies

We start by giving a formal definition of *general dependencies* and show how they capture a broad range of dependency subclasses. In this context, we present the chase procedure by illustrating its application and providing a formal definition.

General Dependencies

According to [12], any dependency d can be expressed as a *general dependency*, i.e as a first-order logic sentence of the form:

$$\forall x_1 \ldots \forall x_n(\phi(x_1, \ldots, x_n) \Rightarrow \exists z_1 \ldots \exists z_k \psi(y_1, \ldots, y_m, z_1, \ldots, z_k))$$

where $\{y_1, \ldots, y_m\} \subseteq \{x_1, \ldots, x_m\}$ and

- ϕ is a potentially empty conjunction of relational atoms, with $VAR(\phi) = \{x_1, \ldots, x_n\}$

- ψ is either
  1. a single equality atom, with $VAR(\psi) \subseteq \{x_1, \ldots, x_n\}$,
     d is then called an *equality generating dependency*, abbreviated as EGD
  2. a non-empty relational atoms conjunction, with $VAR(\psi) = \{y_1, \ldots, y_m, z_1, \ldots, z_k\}$,
     d is then called a *tuple generating dependency*, abbreviated as TGD

  The expressions ψ and ϕ are respectively called the *head* and *body* of d.

Another distinction is made regarding whether or not d contains existential quantifiers: in the former case ($k \geq 1$), d is called *embedded* and, in the later ($k = 0$), d is called *full*. Note that equality generating dependencies are, by nature, always *full*.

In the data integration community, two particular forms of TGDs are especially relevant:

- local-as-view constraints (LAV), expressible as *embedded tuple dependencies* with single relational atom bodies

$$\forall x_1 \ldots \forall x_n (R(x_1, \ldots, x_n) \Rightarrow \exists z_1 \ldots \exists z_k \psi(x_1, \ldots, x_n, z_1, \ldots, z_k))$$

- global-as-view constraints (GAV), expressible as *full tuple dependencies* with single relational atom heads

$$\forall x_1 \ldots \forall x_n (\phi(x_1, \ldots, x_n) \Rightarrow R(x_1, \ldots, x_n))$$

Example 4.3.13. *We exemplify different instances of general dependencies:*

- *the functional dependency $A \hookrightarrow B$ on a relation instance I over the schema $R[A, B]$ is expressible as the* (full) equality generating general dependency*:*

$$\forall v_1 \forall v_2 \forall v_3, (R(v_1, v_2) \wedge R(v_1, v_3) \Rightarrow v_2 = v_3),$$

  where variable $v_1 \in A$ and $\{v_2, v_3\} \subseteq B$

- *the multivalued dependency $C \twoheadrightarrow D$ on a relation instance I over the schema $R[U]$, where $C \cup D \cup E = U$, is expressible as the* full tuple generating general dependency*:*

$$\forall x \forall y_1 \forall y_2 \forall z_1 \forall z_2 (R(x, y_1, z_1) \wedge R(x, y_2, z_2) \Rightarrow R(x, y_1, z_2)),$$

  where variable $x \in C$, $\{y_1, y_2\} \subseteq D$ and $\{z_1, z_2\} \subseteq E$.

As was the case with conjunctive queries, a practical way of representing general dependencies is visually, in tableau form. We illustrate with the previous examples.

A	B
a	b_1
a	b_2
$b_1 = b_2$	

Figure 4.6.: $A \hookrightarrow B$

C	D	E
c	d_1	e_1
c	d_2	e_2
c	d_1	e_2

Figure 4.7.: $C \twoheadrightarrow D$

The Chase Procedure

The *chase* is a fundamental database procedure, originally introduced in [2] and [59], as a tool for testing the *logical implication* of data dependencies. More specificaly, it is used to determine if, for a given set of general dependencies D and for a general dependency d, it holds that $D \models d$. Note that, if D contains only full dependencies, the chase terminates and, if D contains embedded dependencies, it may run indefinitely, albeit providing the right answer, if it stops. An extensive overview of the chase and its application is provided in [67].

Before presenting the chase, let us first define the following preliminary notions, as introduced in [79]. Note that database domains are considered to consist of not only constants, but also variables (labelled nulls), representing unknown values.

Definition 4.3.14 (Symbol Mappings/Homomorphisms[3])**.** *Given symbol sets S and T, a* symbol mapping *$h : S \to T$ is such that, for every symbol $a \in S$, $h(a) \in T$. Note that it is allowed for two distinct symbols $a \neq b$ to be mapped to the same value $h(a) = h(b) \in T$.*

Let R be a relation, I, J relation instances of R over domains $\mathcal{D}_I, \mathcal{D}_J$ and $h : \mathcal{D}_I \to \mathcal{D}_J$, a symbol mapping, such that, for every $c \in \mathcal{D}_I$, $h(c) = c$.

A (symbol) homomorphism *$\bar{h} : I \to J$ is the homomorphic extension of h, such that:*

$$\text{for every } \vec{t} \equiv (t_1, \ldots, t_n) \in I,\ \bar{h}(\vec{t}) = (h(t_1), \ldots, h(t_n)) \in J$$

The homomorphism definition extends naturally for the case of "abstract" database instances, i.e, whose tuples may contain variables.

Tailoring the previous first-order logic definition of general dependencies to the database setting, we introduce the following notations. Let $R[U]$ be a relation schema over a domain $\mathcal{D}$, with $|U| = n$. We denote a generalized dependency γ as:

- if γ is a TGD: $(\vec{t_1}, \ldots, \vec{t_k})/\vec{t}$, where $\vec{t_i}, \vec{t} \in \mathcal{D}^n$ and $\vec{t}$ can contain unique symbols, i.e symbols not among those in t_i (indeed, such is the case for embedded TGDs)
- if γ is an EGD: $(\vec{t_1}, \ldots, \vec{t_k})/x = y$, where x, y are among the symbols in $\vec{t_i}$

[3]The terms *symbol mappings* and *homomorphism* are used interchangably in the database literature.

We abbreviate $\vec{t}$ and, respectively, $x = y$, as $head(\gamma)$, and $\{\vec{t_1}, \ldots, \vec{t_n}\}$, as $body(\gamma)$.

Definition 4.3.15 (General Dependency Satisfiability)**.** *An instance I over $R[U]$ is said to satisfy γ, denoted $I \models \gamma$, if, for any $h : body(\gamma) \to I$:*

- *when $\gamma \equiv (\vec{t_1}, \ldots, \vec{t_k})/\vec{t}$, h can be extended to $\hat{h} : body(\gamma) \cup \{\vec{t}\} \to I$, mapping all unique symbols in $\vec{t}$ to $\hat{h}(\vec{t}) \in I$*
- *when $\gamma \equiv (\vec{t_1}, \ldots, \vec{t_k})/x = y$, $h(x) = h(y)$*

If $I \not\models \gamma$, then there exists a $h : body(\gamma) \to I$, such that:

- *when $\gamma \equiv (\vec{t_1}, \ldots, \vec{t_k})/\vec{t}$, h cannot be extended to $\hat{h} : body(\gamma) \cup \{\vec{t}\} \to I$, mapping all unique symbols in $\vec{t}$ to $\hat{h}(\vec{t}) \in I$*
- *when $\gamma \equiv (\vec{t_1}, \ldots, \vec{t_k})/x = y$, $h(x) \neq h(y)$*

In the case that $I \not\models \gamma$, the pair (γ, h) is called a trigger *for the chase procedure, as it causes it to be applied to I, under homomorphism h, in order to enforce γ.*

Definition 4.3.16 (General Dependency Application)**.** *Assume I to be an instance over $R[U]$. Let us apply a general dependency γ to I, under a homomorphism h, such that (γ, h) is a trigger, i.e, $I \not\models \gamma$. We obtain a new instance J, denoted $I \xrightarrow{(\gamma,h)} J$, such that:*

- ***TGD step**: when $\gamma \equiv (\vec{t_1}, \ldots, \vec{t_k})/\vec{t}$, $J = I \cup \hat{h}(\vec{t})$, where $\hat{h}$ maps **unique symbols** in $\vec{t}$ to **fresh variables***
- ***EGD step**: when $\gamma \equiv (\vec{t_1}, \ldots, \vec{t_k})/x = y$,*
  - *if $h(x)$ and $h(y)$ are both mapped to variables, J is obtained by α-renaming in I all occurences of $h(x)$ to $h(y)$ or vice-versa*
  - *if, among $h(x)$ and $h(y)$, one is a constant and the other a variable, J is obtained by replacing in I all occurences of the variable with the constant*
  - *if h maps x and y to distinct constants, the procedures fails, i.e $I \xrightarrow{(\gamma,h)} \perp$.*

As mentioned in the beginning of the section, in the presence of embedded dependencies, the chase procedure may not terminate, as we illustrate next.

Example 4.3.17 (Chasing with Embedded Dependencies)**.** *Let us consider a database instance $I_0 = \{R(a_1), R(a_2), S(a_1, a_2)\}$ and the embedded TGD*

$$\gamma \equiv \forall x(R(x) \Rightarrow \exists y(R(y) \wedge S(x, y))).$$

We are looking for homomorphisms h, satisfying $R(h(x)) \in I_0$. As such, $h(x) = a_1$ or $h(x) = a_2$. If we choose, for example, $h(x) = a_1$, then we could extend h to $\hat{h}$, where $\hat{h}(y) = a_2$. Hence, (γ, h) would not be a trigger, as $\{R(\hat{h}(y)), S(h(x), \hat{h}(y))\} = \{R(a_2), S(a_1, a_2)\} \subseteq I_0$. Let us examine the case when $h(x) = a_2$. Indeed, (γ, h) is a trigger, since, for any h-extension $\hat{h}$, $S(h(x), \hat{h}(y)) = S(a_2, \hat{h}(y)) \notin I_0$. Applying the

TGD chase step, we obtain $I_0 \xrightarrow{(\gamma,h)} I_1$*, where* $I_1 = I_0 \cup \{R(\mathbf{a_3}), S(a_2, \mathbf{a_3})\}$ *and* $\mathbf{a_3}$ *is a fresh variable. Since,* $I_1 \not\models \gamma$*, we are forced to continue iterating the procedure and, by a similiar reasoning, we obtain* $I_1 \xrightarrow{(\gamma,h)} I_2$*, where* $I_2 = I_1 \cup \{R(\mathbf{a_4}), S(a_3, \mathbf{a_4})\}$ *and* $\mathbf{a_4}$ *is a fresh variable. We conclude the chase is nonterminating.*

Informally, the chase starts from the instance represented by the tableau part of d, i.e assuming that the body of d is satisfied. The procedure consists in iterating *chase steps* that apply dependencies in D, as described in Definition 4.3.16. If the applied dependency is a TGD, a new tuple is added and, if it is an EGD, the instance is modified accordingly. If no dependency can be further applied, we can check whether the head of d is satisfied in the resulting instance. Specifically, if d is a TGD, we check whether the instance contains a tuple that agrees with it, modulo renaming. If d is an EGD, we check whether the equated symbols have been mapped to the same value.

Example 4.3.18. *Given the dependencies* γ_1, γ_2 *and* γ_3*, presented in tableau form in Figure 4.8, let us apply the chase procedure to determine whether* $\{\gamma_1, \gamma_2\} \models \gamma_3$*.*
We start from the instance (i)*, the tableau part of* γ_3*.*

- ***TGD step****: Applying* γ_1 *to* (i) *consists in finding a homomorphism* h*, where* $\{h(a_1, b_1, c_1, d_1), h(a_1, b_2, c_2, d_2)\} \subseteq (i)$ *and* (γ_1, h) *are a* trigger *for the chase, i.e there is no* h*-extension* $\hat{h}$*, such that* $\hat{h}(a_1, b_1, c_2, d_3) \in (i)$*.*

  Indeed, one such mapping is:
  $h(a_1) = a_4, h(b_1) = b_5, h(c_1) = c_6, h(d_1) = d_7, h(b_2) = b_4, h(c_2) = c_5, h(d_2) = d_6$*,*
  since $\{h(a_1, b_1, c_1, d_1), h(a_1, b_2, c_2, d_2)\} = (i)$ *and, for any* h*-extension* $\hat{h}$*,*
  $\hat{h}(a_1, b_1, c_2, d_3) = (a_4, b_5, c_5, \hat{h}(d_3)) \notin (i)$*.*

  Consequentely, $I \xrightarrow{(\gamma_1,h)} (ii)$*, where* $(ii) = (i) \cup \{\hat{h}(a_1, b_1, c_2, d_3)\}$*.*
  Note that, as the existentially quantified d_3 *is* ***unique*** *in* γ_1*,* $\hat{h}(d_3)$ *is assigned to be the* ***fresh variable*** d_8*. Hence,* $\hat{h}$ *corresponds to:* $\hat{h}(a_1) = a_4$*,* $\hat{h}(b_1) = b_5$*,*
  $\hat{h}(c_1) = c_6, \hat{h}(d_1) = d_7, \hat{h}(b_2) = b_4, \hat{h}(c_2) = c_5, \hat{h}(d_2) = d_6$ *and* $\hat{h}(d_3) = d_8$*.*

- ***EGD step****: Applying* γ_2 *to* (ii) *makes* d_7 *and* d_8 *equal in* (iii)*. Also, as* b_6 *is existentially quantified in* γ_3*, it can be instantiated by* b_5*.*

Since the tuple generated in γ_3 *occurs in* (iii)*,* γ_3 *is logically implied by* γ_1 *and* γ_2*.*

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_1	b_1	c_2	$\mathbf{d_3}$

tgd γ_1

A	B	C	D
a_2	b_3	c_3	d_4
a_3	b_3	c_4	d_5
	d_4	$=$	d_5

egd γ_2

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_6	c_5	d_7

tgd γ_3

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7

(i)

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_5	c_5	$\mathbf{d_8}$

(ii)

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	$\mathbf{d_7}$
a_4	$\mathbf{b_5}$	c_5	$\mathbf{d_7}$

(iii)

Figure 4.8.: Applying Dependencies

5. Standard Datalog

In this chapter, we introduce the standard DATALOG language, a logic based data model, that supports recursion. We outline its syntax in Section 5.1 and two-equivalent semantics: the minimal model semantics, in Section 5.2.1, and the fixpoint semantics, in Section 5.2.2. In our presentation we will specialize the base definitions regarding the syntax and semantics of first-order languages, given in Chapter 3.

5.1. Syntax

Being a subset of Prolog, DATALOG is rooted in logic programming, which, in turn, is a rule-based formalism. As such, DATALOG programs or knowledge-bases are represented as finite sets of *clauses* that are either *facts*, i.e assertions known to be true, or *rules*, i.e sentences that allow the inference of new facts from existing ones.

Definition 5.1.1 (Symbols)**.** 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.*

A DATALOG expression e is either a term t, an atom A, a clause C or a program P. We define each of these objects according to [58].

Definition 5.1.2 (Terms)**.** *A* term t *is either a constant or a variable.*

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

Remark 5.1.3. *We denote predicate, variable and term sequences using the vector notation:* $\vec{p} = p_1, p_2, \ldots, p_n$*,* $\vec{x} = x_1, x_2, \ldots, x_n$ *and* $\vec{t} = t_1, t_2, \ldots, t_n$*, where* n *is the sequence length denoted as* $|\cdot|$*, i.e* $|\vec{p}| = |\vec{x}| = |\vec{t}| = n$*.*

Definition 5.1.4 (Atoms)**.** *Let* p *be a predicate with* $ar(p) = n$ *and* $\vec{t}$ *a term sequence with* $|\vec{t}| = n$*. An* atom A *is an expression of the form :*

$$A ::= p(\vec{t}), \text{ where } p \in \mathcal{P}$$

The terms $t_1, t_2, \ldots, t_n$ *are called its* arguments*. We denote* $sym(A) = p$*.*

Definition 5.1.5 (Clauses)**.** *The general syntax of a* DATALOG *clause* C *is :*

$$C ::= A_0 \leftarrow A_1, \ldots, A_m, \text{ or, in an alternative notation, } C ::= A_0 :- A_1, A_2 \ldots, A_m$$

Remark 5.1.6. *Note that the clause* C *represents :*

- *a* fact*, when* $m = 0$*, i.e* $C \equiv A_0 \leftarrow$
- *a* query*, when* $m \geq 1$ *and there is no head atom, i.e* $C \equiv \quad \leftarrow A_1, A_2 \ldots, A_m$
- *a* rule*, otherwise*

The atom A_0 *is called the* head *of* C *and the atom list* $A_1, A_2, \ldots, A_m$*, its* body*. As such, from now on, we will denote* A_0 *as* H *and* $A_1, A_2, \ldots, A_m$ *as* $B_1, B_2, \ldots, B_m$*. Informally,* C *translates into "****if*** B_1 ***and*** $B_1 \ldots$ ***and*** B_m ***then*** H*".*

Definition 5.1.7 (Programs)**.** *A* program P *is a finite set of clauses.*

$$P ::= C_0, \ldots, C_k, \text{ where } k \in \mathbb{N}$$

Note that the commas denote conjunction.

Definition 5.1.8 (Ground Expressions)**.** *Variable-free expressions are called* ground*. We denote ground atoms, clauses and programs as* $\bar{A}, \bar{C}$ *and* $\bar{P}$*.*

To ensure only a **finite** number of **ground** facts can be derived from a standard DATALOG program, a syntactic restriction is imposed on its clauses.

Definition 5.1.9 (Safety Condition)**.** *Let* P *be program and* C *one of its clauses.* ***All variables in the head of*** C ***should also appear in its body****. As a corollary of this,* ***all facts in*** P ***have to be ground****.*

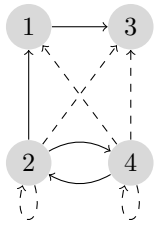
Extensional and Intensional Predicates Predicates of a DATALOG program can be divided into: *extensional/base predicates*, if they occur as clause heads in unit clauses, and *intensional/derived predicates*, otherwise. From a database perspective, the former correspond to *stored relations* and the latter, to *virtual relations (views)*. As such, a DATALOG program P can be seen as the union of two disjoint databases: an *extensional database* $edb(P)$, i.e the set of ground facts (whose predicates are all extensional), and an *intensional database* $idb(P)$, i.e the set of rules (whose head predicates are all intensional). The corresponding program schema is $sch(P) = edb(P) \cup idb(P)$. From a logic programming standpoint this distinction is not consequential. However, in practice it serves the pragmatic purpose of decoupling the (typically much larger) extensional component from the intensional one, which can thus be independently pre-processed. The implications of the intensional versus extensional distinction in the deductive database setting were originally described in [74].

Recursive Rules Given that a head literal can also appear either in the body of the same clause or in that of other program clauses, DATALOG rules and programs have the property of being *recursive*. In this respect, DATALOG extends the expressivity of relational calculus, in which *transitive closure* queries cannot be stated. Recall that the *transitive closure* of a relation R is the smallest relation R^* such that 1) $R \subseteq R^*$ and 2) R^* is transitive.

Example 5.1.10. *We illustrate this with a* Datalog *program computing the transitive closure of the graph below. The graph's configuration is encoded as a series of* facts *corresponding to existing edges. In particular, the transitive closure corresponding to the edge relation* `e` *is modeled by the above rules defining* `tc`*.*

```
e(1, 3).
e(2, 1).
e(4, 2).
e(2, 4).

tc(X, Y) :- e(X, Y).
tc(X, Y) :- e(X, Z), tc(Z, Y).
```



Example 5.1.11. Datalog *can also serve as a basis for expressing various types of access control policies. The present example is taken from [34], in which the following policy for controlling access to conference review scores is considered:*

1. *During the submission phase, an author may submit a paper.*
2. *During the review phase, reviewer r may submit a review for paper p if r is assigned to review p.*
3. *During the meeting phase, reviewer r can read the scores for paper p if r has submitted a review for p.*
4. *Authors may never read scores.*

Let Subjects, Actions and Rules be sorts and Σ be a first-order relational signature, containing the ternary predicates `Permit` *and* `Deny` *over Subjects $\times$ Actions $\times$ Rules. A policy over Σ is expressed as a* Datalog *program P, where $\{$`Permit`, `Deny`$\} \subseteq idb(P)$.*

Based on the XACML *standard (see [69])*[1]*, the policy above can be modeled as:*

```
Permit(a, submit_paper, p) :- author(a), paper(p), phase(submission)

Permit(r, submit_review, p) :-
  reviewer(r), paper(p), assigned(r,p), phase(review)

Permit(r, read_scores, p) :-
  reviewer(r), paper(p), has_reviewed(r,p), phase(meeting)

Deny(a, read_scores, p) :- author(a), paper(p)
```

[1] which explicitly uses the `Deny` predicate instead of the negation of `Permit`

5.2. Semantics

As originally defined in [81], DATALOG has two *equivalent* semantics: one based on minimal models and the other, on the least-fixpoint.

5.2.1. Minimal Model Semantics

DATALOG programs have a natural translation into first-order logic theories, as follows.

First-Order Logic Translation

- **Atoms.** An atom A can be represented as a first-order logic formula A^*
- **Clauses.** A clause $C \equiv H \leftarrow B_1, \ldots, B_m$ translates into:

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

  with $X_i \in VAR(H), \forall i \in [1,q]$; $X_j \notin VAR(H)$; $X_j \in \bigcup_{k \in [1,m]} VAR(B_k), \forall j \in [q+1,l]$.

- **Programs.** A program P can be identified with a set of first-order logic formulas, corresponding to $P^* = \bigwedge_{C \in P} C^*$.

Remark 5.2.1. DATALOG *clauses* $C \equiv H \leftarrow B_1, \ldots, B_m$ *are equivalent to:*

- full tuple generating dependencies *(see Section 4.3.3)*

$$C^* \Leftrightarrow \forall X_1 \ldots \forall X_l (B_1 \wedge \ldots \wedge B_m \Rightarrow H), \text{ with } X_k \in VAR(C), \forall k \in [1,l].$$

- definite Horn clauses *(see Definition 3.3.4) without function symbols*

$$C^* \Leftrightarrow \forall X_1 \ldots \forall X_l (\neg B_1 \vee \ldots \vee \neg B_m \vee H).$$

Example 5.2.2 (Graph Transitive Closure)**.** *The first-order logic translation of the* DATALOG *program above is given by the formula:*

$$e(1,3) \wedge e(2,1) \wedge e(4,2) \wedge e(2,4) \wedge (\forall X \forall Y (e(X,Y) \Rightarrow tc(X,Y))) \wedge (\forall X \forall Y \forall Z (e(X,Z) \wedge tc(Z,Y) \Rightarrow tc(X,Y)))$$

Having established DATALOG to be a fragment of first-order language, we proceed to giving its interpretation, as a particular instance of that given in Section 3.

Let us fix a program P, with the symbol signature $\Sigma = (\mathcal{C}, \mathcal{P}, ar)$.

A Σ-*structure* $\mathcal{I}$ for P is given by $\mathcal{I} = (U,\ I : \Sigma \to U \cup \{\top, \bot\})$, i.e, a non-empty universe U and an interpretation function I mapping Σ to $\{\top, \bot\}$, such that:

- for every $c \in \mathcal{C}$: $c^I \in U$
- for every $p \in \mathcal{P}$, where $ar(p) = n$: p^I is a mapping $p^I : U^n \to \{\top, \bot\}$

Interpretations For 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:

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

Models An interpretation I is a *model* for

- a clause $H \leftarrow B_1, \ldots, B_n$, if $[\![H \leftarrow B_1, \ldots, B_n]\!]^I = \top$
- a program P, if $[\![C]\!]^I = \top$, for all $C \in P$

We use the standard notation $I \models C$ and $I \models P$, where C and P, are said to *satisfy* I.

Logical Consequence A fact F is a *logical consequence* of a P, i.e $P \models F$, iff

$$I \models P \text{ implies } I \models F, \text{ for all interpretations } I$$

The set of all logical consequences of a program P is denoted as $\mathbf{cons}(P)$.

Definition 5.2.3. *The fact that a fact F is a logical consequence of a program P can be expressed based on the* Herbrand Semantics *(see Section 3.2) of P, consisting of the following:*

- Herbrand Universe*: set of all program constants, denoted $adom(P)$.*
- Herbrand Base*: set of all ground atoms that can be built from predicates $p \in \mathcal{P}$ and constants in $adom(P)$, denoted $\mathbb{B}_P$.*
- Herbrand Interpretation $I_{\mathcal{H}}$*: subset of the Herbrand base $\mathbb{B}_P$.*

As such, it holds that: $P \models F$ iff $I_{\mathcal{H}} \models P$ implies $F \in I_{\mathcal{H}}$[2]*.*

[2]Since $I_{\mathcal{H}} \models F$ iff $F \in I_{\mathcal{H}}$.

Restricting ourselves to the Herbrand semantics setting, we thus define the grounding of P with a valuation $\iota : BV(P) \to adom(P)$ as $\iota P \equiv \bigcup_{C \in P} \iota C$, where the grounding of a clause $C \equiv p_0(\vec{t_0}) \leftarrow p_1(\vec{t_1}), \ldots, p_m(\vec{t_m})$ is $\iota C \equiv p_0(\iota\vec{t_0}) \leftarrow p_1(\iota\vec{t_1}), \ldots, p_m(\iota\vec{t_m})$.

For a Herbrand interpretation $I_{\mathcal{H}}$, it holds that:

- $I_{\mathcal{H}} \models \iota C$ iff $\{p_1(\iota\vec{t_1}), \ldots, p_m(\iota\vec{t_m})\} \subseteq I_{\mathcal{H}}$ implies $p_0(\iota\vec{t_0}) \in I_{\mathcal{H}}$
- $I_{\mathcal{H}}$ is a **Herbrand model** of P iff $I_{\mathcal{H}} \models \iota P$, for all $\iota : BV(P) \to adom(P)$

Note that, as illustrated next, a definite program can have multiple Herbrand models.

Example 5.2.4. *Consider the program* $\tilde{P} = \{p(a), p(b), q(a), r(X) \leftarrow p(X), q(X)\}$.

- $adom(P) = \{a, b\}$ *and* $\mathbb{B}_P = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$
- $I_1 = \{p(a), p(b), q(a)\}$, $I_2 = \{p(a), p(b), q(a), r(a)\}$, $I_3 = \{p(a), p(b), q(a), r(a), r(b)\}$, $I_4 = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$ *and* $I_5 = \emptyset$ *are* Herbrand interpretations *of* P, *but only* I_2, I_3 *and* I_4 *are* Herbrand models *of* P.

Theorem 5.2.5 (Model Intersection Property)**.** *If* M_1, M_2 *are Herbrand models of a definite program* P, *then* $M_1 \cap M_2$ *is also a model of* P.

Minimal Model Semantics Under the partial ordering induced by set inclusion, a Herbrand model can be defined as being *minimal*, if none of its proper subsets are also models. For example, the model I_2 above is minimal. As a consequence of the model intersection property, any **definite** program P has an ***unique minimal model*** $\mathbf{M}(P)$ that is the intersection of all its Herbrand models.

This is the **intended semantics** of P, i.e $\mathbf{cons}(P) = \mathbf{M}(P)$.
The reasons for this choice are discussed in Chapter 12 from [1].

5.2.2. Fixpoint Semantics

The *fixpoint semantics* is a *denotational* semantics based on the idea that definite program models can be seen as pre-fixed points of a special closure operator named the *immediate consequence operator*. The *least-fixpoint semantics* bridges the *declarative* minimal-model semantics presented above to the *procedural* bottom-up evaluation.

We first introduce basic fixpoint theory definitions.

Definition 5.2.6 (Complete Lattices)**.** *Let us define a* complete lattice *as* $\langle \mathcal{L}, \subseteq \rangle$, *where* $\mathcal{L}$ *is an ordered set with respect to inclusion and any set* $A \subset \mathcal{L}$ *has a* greatest lower bound $\bigcap A$ *and a* lowest upper bound $\bigcup A$.

Definition 5.2.7 (Complete Lattice Operator Properties)**.** *An operator* $T : \mathcal{L} \to \mathcal{L}$ *:*

- *is* monotonic, *if* $I_1 \subseteq I_2$ *implies* $T(I_1) \subseteq T(I_2)$, *for all* $I_1, I_2 \subseteq \mathcal{L}$

- *has a* pre-fixed point I, *if* $T(I) \subseteq I$
- *has a* fixpoint I, *if* $T(I) = I$

The following theorem by [78] is key to the fixpoint semantics given in [81].

Theorem 5.2.8 (Knaster-Tarski Theorem)**.** *Let $\langle \mathcal{L}, \subseteq \rangle$ be a complete lattice and the operator T on $\mathcal{L}$. If T is monotonic, then it has a least fixpoint.*

This result is applicable in our setting, if we consider the complete lattice of Herbrand interpretations $\langle \mathcal{P}(\mathbb{B}_P), \subseteq \rangle$ and the operator $T_P : \mathcal{P}(\mathbb{B}_P) \to \mathcal{P}(\mathbb{B}_P)$, introduced below.

Definition 5.2.9 (Immediate Consequence Operator)**.** *The* immediate consequence operator T_P *for a program P operates on program interpretations, i.e, $T_P : \mathcal{P}(\mathbb{B}_P) \to \mathcal{P}(\mathbb{B}_P)$, such that, for a set of ground atoms $I \subset \mathbb{B}_P$:*

$$T_P(I) = \{F \in \mathbb{B}_P \mid F \in I \vee F = head(\iota C), C \in P \wedge body(\iota C) \subseteq I\}$$

The ground atom F is called an immediate consequence *of the program P.*

Remark 5.2.10. *Note that the immediate consequence operator is inherently* inflationary[3]*, i.e $I \subseteq T_P(I)$, for all program interpretations I.*

Since the immediate consequence operator is *monotonous*, according to Theorem 5.2.8, it has a least fixpoint, $lfp(T_P)$. This is computed by iterating T_P, starting from the empty interpretation, as shown next.

Definition 5.2.11 (Powers/Iteration of the Immediate Consequence Operator)**.** *The powers of the immediate consequence operator are given by:*

$$T_P \uparrow 0 = \emptyset$$
$$T_P \uparrow (n+1) = T_P(T_P \uparrow n)$$
$$T_P \uparrow \omega = \bigcup_{n \geq 0} T_P \uparrow n$$

There exists some ω such that $T_P \uparrow \omega = lfp(T_P)$.

Example 5.2.12. *Revisiting the transitive closure example from Figure 6.1.1, we have:*

$$T_P \uparrow 0 = \emptyset$$
$$T_P \uparrow 1 = T_P(\emptyset) = \{e(1,3), e(2,1), e(4,2), e(2,4)\}$$
$$T_P \uparrow 2 = T_P(T_P \uparrow 1) = T_P \uparrow 1 \cup \{tc(1,3), tc(2,1), tc(4,2), tc(2,4)\}$$
$$T_P \uparrow 3 = T_P(T_P \uparrow 2) = T_P \uparrow 2 \cup \{tc(2,3), tc(4,1), tc(4,4), tc(2,2)\}$$
$$T_P \uparrow 4 = T_P(T_P \uparrow 3) = T_P \uparrow 3 \cup \{tc(4,3)\}$$
$$T_P \uparrow 5 = T_P(T_P \uparrow 4) = T_P \uparrow 4$$

Hence, $T_P \uparrow \omega = T_P \uparrow 4$.

It follows that $lfp(T_P) = \{tc(1,3), tc(2,1), tc(4,2), tc(2,4), tc(2,3), tc(4,1), tc(4,4), tc(2,2), tc(4,3)\}$.

[3] This property is also sometimes called *supportedness*

Lemma 5.2.13. *For a definite program P and a Herbrand Structure $\mathcal{H} = (U_\mathcal{H}, I)$:*

$$I \text{ is a pre-fixed point of } T_P, \text{ i.e } T_P(I) \subseteq I \Leftrightarrow I \models P$$

Proof. Recall that $I \models P \Leftrightarrow I \models \iota P$, for all valuations $\iota : BV(P) \to U_\mathcal{H}$.

$\Rightarrow$ Let $\bar{H} \leftarrow \bar{B_1}, \ldots, \bar{B_n} \in \iota P$. If $\{\bar{B_1}, \ldots, \bar{B_n}\} \subseteq I$, then, by definition of T_P, $\bar{H} \in T_P(I)$. Since $T_P(I) \subseteq I$, then $\bar{H} \in I$. Hence, $I \models \bar{H} \leftarrow \bar{B_1}, \ldots, \bar{B_n}$.
It follows that $I \models P$.

$\Leftarrow$ Let $\bar{H} \in T_P(I)$. Then, there exists a ground instance $\bar{H} \leftarrow \bar{B_1}, \ldots, \bar{B_n}$ of a clause in P, such that $\{\bar{B_1}, \ldots, \bar{B_n}\} \subseteq I$. From the hypothesis, $\bar{H} \in I$. Thus, $T_P(I) \subseteq I$.

□

The following theorem relates the immediate fixpoint operator to the unique minimal model from Section 5.2.1.

Theorem 5.2.14 (van Emden and Kowalski)**.** *The unique minimal Herbrand Model* $\mathbf{M}(P)$ *of a definite program P is* $\mathbf{M}(P) = lfp(P) = T_P \uparrow \omega$.

Operationally, the action of the immediate consequence operator is captured by the following inference rule, called the Elementary Production Principle (EPP) (see [19])

$$\frac{H \leftarrow B_1, \ldots, B_n \qquad \{F_1 \ldots F_n\} \subseteq I \qquad \exists \theta,\ \theta B_1 = F_1 \ \wedge\ \ldots\ \wedge\ \theta B_n = F_n}{\theta H}\ \mathsf{EPP}$$

Figure 5.1.: Elementary Production Principle for Clausal Logical Consequence Inference

Note that the EPP rule describes, in fact, hyperresolution (see Section 3), i.e the iterated application of binary resolution steps, as detailed in Figure 5.2.

Definition 5.2.15 (Fact Inference)**.** *A fact F can be inferred from a definite program P, according to the following rules:*

$$\cfrac{\cfrac{\cfrac{\cfrac{H \vee \neg B_1 \vee \ldots \vee \neg B_n \qquad F_1 \qquad \exists \theta_1.\theta_1 B_1 = F_1 \qquad \theta_1 \preceq \theta}{\theta_1 H \vee \neg \theta_1 B_2 \vee \ldots \neg \theta_1 B_n \qquad F_2 \qquad \exists \theta_2.\theta_2\theta_1 B_2 = F_2 \qquad \theta_2\theta_1 \preceq \theta}}{\theta_2\theta_1 H \vee \neg\theta_2\theta_1 B_3 \vee \ldots \vee \neg\theta_2\theta_1 B_n \qquad F_3 \qquad \exists \theta_3.\theta_3\theta_2\theta_1 B_3 = F_3 \qquad \theta_3\theta_2\theta_1 \preceq \theta}}{\vdots}}{\theta_{n-1}\ldots\theta_1 H \vee \neg\theta_{n-1}\ldots\theta_1 B_n \qquad F_n \qquad \exists \theta_n.\theta_n \ldots \theta_1 B_n = F_n \qquad \theta_n \ldots \theta_1 \preceq \theta}$$

$$\theta_n \ldots \theta_1 H$$

Figure 5.2.: Logical Consequence Inference as Iterated Resolution

For a DATALOG program P, the set of all logical $\mathbf{cons}(P)$ is obtained by iterating the EPP rule, until reaching saturation, as illustrated by the pseudo-code:

```
1: procedure CONS(P)
2:     X_0 ← EDB(P)
3:     i ← 0
4:     while X_i ≠ X_{i+1} do
5:         for all C ≡ H ← L_1, ..., L_n ∈ P do
6:             X_i ← X_i ∪ EPP(C, X_i)
7:             i ← i + 1
8:     return X_i
```

Theorem 5.2.16. *The* CONS *algorithm is sound and complete.*

Proof. Corollary of the analogous properties of hyperresolution established in [76]. □

6. Datalog with Negation

In this chapter, we present Datalog with negation, a language strictly more expressive than the relational algebra and the safe relational calculus, reviewed in Chapter 4. First, we extend standard Datalog programs with the negation operator in Section 6.1. Then, we give the stratified semantics for such programs in Section 6.2. Finally, we briefly overview alternative semantics in Section 6.3.

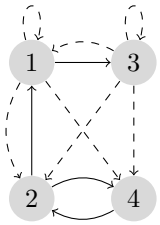
6.1. Syntax

Revisiting Example 5.1.10, suppose we wish to compute the pairs of disconnected graph nodes, i.e the *complemented transitive closure*, marked by the set of dashed arcs in the example below. To define the corresponding `ctc` predicate, we need to use *negation*.

Example 6.1.1. *As such, in a preliminary attempt, we could state that* `ctc` *holds for all pairs not in the transitive closure computed by* `tc`*.*

```
e(1, 3).
e(2, 1).
e(4, 2).
e(2, 4).

tc(X, Y) :- e(X, Y).
tc(X, Y) :- e(X, Z), tc(Z, Y).
ctc(X, Y) :- not tc(X, Y).
```



At the language level, adding negation amounts to extending the syntax of standard Datalog defined in Section 5.1. This implies introducing a new primitive for *literals* and adjusting the definition for clauses correspondingly.

Definition 6.1.2 (Literals)**.** *A literal L is either a positive or a negated atom:*

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

Definition 6.1.3 (Clauses)**.** *A clause C has a positive atom head and a body of literals:*

$$C ::= A \leftarrow L_1, \ldots, L_n$$

We denote the sets of positive and negative atoms in the body of C as $body^+(C)$ and $body^-(C)$.

Safety The safety condition imposed on standard Datalog clauses - see Section 5.1.9 - fails at ensuring the finiteness of a model that we could "naively" compute in this setting, following the semantics from Section 5.2. Indeed, while the definition of `ctc` obeys the condition, an infinity of facts are not in the transitive closure computed in Example 5.2.12. The ensuing extended restriction is standard in the literature.

Definition 6.1.4 (Extended Safety Condition)**.** *Let P be program and C one of its clauses.* ***All variables from the negated atoms in the body of*** *C* ***should also appear among the arguments of its positive atoms****, i.e* $V(body^-(C)) \subseteq V(body^+(C))$.

Example 6.1.5. *The following* ***safe*** *program is an alternative to that in Example 6.1.1.*

```
e(1, 3).
e(2, 1).
e(4, 2).
e(2, 4).

v(X) :- e(X, Y).
v(Y) :- e(X, Y).
tc(X, Y) :- e(X, Y).
tc(X, Y) :- e(X, Z), tc(Z, Y).
ctc(X, Y) :- v(X), v(Y), not tc(X, Y).
```

It is important to note that this condition is unnecessary if the database domain consists of constants from the extensional part, i.e from base facts. Consequently, the formalization developed in Chapter 9 does away with it, establishing that we indeed can - *without loss of generality* - restrict ourselves to the *active domain*.

6.2. Stratified Semantics

We present the stratified semantics of Datalog programs with negation, by first discussing the restricted case of semipositive programs. We then introduce the revised definition of logical consequence and that of a program stratification and, based on these, we compute the iterative least fixpoint model of a stratified program. We conclude by stating the main property characterizing stratified semantics.

6.2.1. Semipositive Datalog

Semipositive Datalog programs are programs in which negation is only applied to atoms with *edb* predicates. In this resticted case, the negated atoms can be replaced by their complement with respect to the program's Herbrand base. Consequently, their semantics can be defined the same way as that for standard Datalog programs (see Chapter 5).

Example 6.2.1. *The program in Example 6.1.5 is not* semipositive*, as the negated predicate* `tc` *is intensional. However, it is equivalent to the program below:*

```
v(1). v(2). v(3). v(4).
tc(1, 3). tc(2, 1). tc(4, 2). tc(2, 4).
tc(2, 3). tc(4, 1). tc(4, 4). tc(2, 2). tc(4, 3).

v(X) :- e(X, Y).
v(Y) :- e(X, Y).
ctc(X, Y) :- v(X), v(Y), not tc(X, Y).
```

which, in turn, is equivalent to following standard Datalog program: This is obtained by

```
v(1). v(2). v(3). v(4).
tc'(1, 1). tc'(1, 2). tc'(1, 4).
tc'(3, 1). tc'(3, 2). tc'(3, 3). tc'(3, 4).

v(X) :- e(X, Y).
v(Y) :- e(X, Y).
ctc(X, Y) :- v(X), v(Y), tc'(X, Y).
```

replacing `tc` *with its complement* `tc'` *with respect to the Hebrand base of the program.*

6.2.2. Logical Consequence

Let us now consider negation in a more general setting than that from the previous section, i.e, when also *idb* predicates can appear negated. To this end, the immediate consequence operator from Definition 5.2.9 requires amendement.

Definition 6.2.2 (Extended Immediate Consequence Operator)**.** *Let P be a program with negation. The* extended immediate consequence operator $\tilde{T}_P$ *operates on program interpretations, i.e, $\tilde{T}_P : \mathcal{P}(\mathbb{B}_P) \to \mathcal{P}(\mathbb{B}_P)$, such that, for a set of ground atoms $I \subset \mathbb{B}_P$:*

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

Note that, compared to the operator from Definition 5.2.9, the extended immediate operator is not inherently inflationary. Specifically, it is not the case that $I \subseteq \tilde{T}_P(I)$, for all unrestricted[1] instances I of P. This impacts monotonicity as explained next.

Indeed, despite clause syntax or database domain constraints, several issues arise when defining an appropriate semantics for Datalog programs with negation. As we illustrate below, relying on the semantics from Section 5 fails on numerous accounts.

[1] i.e instances over the full schema of P, given by the set of its head symbols

Example 6.2.3. *Consider the program* $P = \{p \leftarrow \neg q\}$.

- $I_1 = \{p\}$ *and* $I_2 = \{q\}$ *are both* minimal Herbrand models. (uniqueness violated)
- $I_1 \cap I_2 = \emptyset \not\models P$. (model interesection property violated)
- $\tilde{T}_P(\emptyset) = \{p\}$ *and* $\tilde{T}_P(\{q\}) = \emptyset$ (monotonicity violated)

Moreover, the iteration of the consequence operator also has a problematic behaviour.

Example 6.2.4. *Consider the program* $\tilde{P} = \{p \leftarrow \neg q, q \leftarrow \neg p\}$. *Computing the iteration of the extended consequence operator according to Definition 5.2.11, we obtain results alternating between* $\emptyset$ *and* $\{p, q\}$.

As a remedy, [3] develops a theory of non-monotonic operators and, based on it, defines a **stratified semantics** for Datalog programs with negation. A key point is the below definition of iteration that, in being inflationary, ensures monotonicity. Note that this is different from the inflationary fixpoint semantics, which is based on an alternative definition of the extended immediate consequence operator. This distinction is detailed in [1].

Definition 6.2.5 (Powers/Iteration of Non-monotonic Operators)**.** *Let* $\langle \mathcal{L}, \subseteq \rangle$ *be a complete lattice and* $T : \mathcal{L} \to \mathcal{L}$, *a nonmonotonic operator defined on it. Its powers are:*

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

Note that this subsumes the monotonic operator iteration from Definition 5.2.11, i.e, $T \uparrow (n+1) = T(T \uparrow n)$.

Remark 6.2.6. *Applying the previous definition to Example 6.2.4, it follows that* $\{p, q\}$ *is the fixpoint model for* $\tilde{P}$.

6.2.3. Program Stratifications

Having introduced a means of iterating the non-monotonic extended consequence operator from Definition 6.2.2, we proceed to defining what a program *stratification* is and how to compute it. For this, we first need the notion of a *predicate definition*.

Definition 6.2.7 (Predicate Definitions)**.** *Let* P *be a program* P *and* $C \in P$. *The definition of a program predicate* p *is given by:*

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

Definition 6.2.8 (Stratified Programs)**.** *Let* $\sigma : \mathcal{P} \to [1, n]$ *be a mapping indexing the predicate symbols of a program* P, *such that, for any clause* $C \in P$, *where*

$$C \equiv H \leftarrow B_1, \ldots, B_k, \neg C_1, \ldots, \neg C_l$$

the following properties hold:

- $\sigma(B_i) \leq \sigma(H)$, *for all* $i \in [1, k]$
- $\sigma(C_i) < \sigma(H)$, *for all* $i \in [1, l]$

We call σ *a* stratification *and, for* $i \in [1, n]$, *we call each* P_i *a* stratum, *where*

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

The mapping σ *induces a partitioning of* P *into* $P = P_1 \sqcup \ldots \sqcup P_n$ [2], *such that, for every predicate* $p \in \mathcal{P}$ *and clause* $C \in P_i$, *it holds that:*

- *if* $p \in \bigcup_{L \in body^+(C)} sym(L) \Rightarrow def(p) \subseteq \bigcup_{j \leq i} P_j$
- *if* $p \in \bigcup_{L \in body^-(C)} sym(L) \Rightarrow def(p) \subseteq \bigcup_{j < i} P_j$

A stratification *of* P *- according to* σ *- is* $\{P_1, \ldots, P_n\}$, *denoted as* $\bar{P_n}$. *We call each* P_i *a* program slice.

Remark 6.2.9. *A program can have multiple stratifications, as seen in Example 6.2.12.*

Remark 6.2.10. *Each program slice* P_i *is a* semipositive *Datalog program relative to previously defined relations. This follows from the second stratification property.*

In [80], the following algorithm is given for testing if a program is stratifiable and, if so, for computing one of its stratifications:

```
1:  procedure STRATIFICATION(P)
2:      for all predicates p in P do
3:          stratum[p] ← 1
4:      repeat
5:          for all clause C in P with head predicate p do
6:              for all negated subgoal of C with predicate q do
7:                  stratum[p] ← max(stratum[p], 1 + stratum[q])
8:              for all nonnegated subgoal of C with predicate q do
9:                  stratum[p] ← max(stratum[p], stratum[q])
10:     until no stratum changes or a stratum exceeds the predicate count in P
```

Based on this algorithm, we illustrate program stratification with the examples below.

Example 6.2.11. *The programs* $\{p \leftarrow \neg p\}$ *and* $\{p \leftarrow q, q \leftarrow \neg p\}$ *are not stratified.*

[2] where $\sqcup$ denotes the pairwise disjoint set union

Example 6.2.12. *Consider the program P consisting of the following clauses:*

$$\begin{array}{lll} C_1 = & p(X) \leftarrow & \neg q(X), r(X) \\ C_2 = & p(X) \leftarrow & \neg t(X), q(X) \\ C_3 = & q(X) \leftarrow & s(X), \neg t(X) \\ C_4 = & r(X) \leftarrow & t(X) \\ C_5 = & q(a) \leftarrow & \\ C_6 = & s(b) \leftarrow & \\ C_7 = & t(a) \leftarrow & \end{array}$$

After iterating the stratification algorithm twice, each predicate symbol in P is associated to a stratum number, as illustrated in the last line of the table below :

	p	q	r	s	t
	1	1	1	1	1
C_1	2	1	1	1	1
C_2	2	1	1	1	1
C_3	2	2	1	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_7	2	2	1	1	1
C_1	3	2	1	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_7	**3**	**2**	**1**	**1**	**1**

Hence, P_3 can be partitioned into $P_3 = P_1 \sqcup P_2 \sqcup P_3$, where:

$$P_1 = \left\{ \begin{array}{l} r(X) \leftarrow t(X) \\ s(b) \leftarrow \\ t(a) \leftarrow \end{array} \right\},\ P_2 = \left\{ \begin{array}{l} q(X) \leftarrow s(X), \neg t(X) \\ q(a) \leftarrow \end{array} \right\},\ P_3 = \left\{ \begin{array}{l} p(X) \leftarrow \neg q(X), r(X) \\ p(X) \leftarrow \neg t(X), q(X) \end{array} \right\}$$

Other types of partitions can also be given:

$$P_1 = \left\{ \begin{array}{l} r(X) \leftarrow t(X) \\ t(a) \leftarrow \end{array} \right\},\ P_2 = \left\{ \begin{array}{l} q(X) \leftarrow s(X), \neg t(X) \\ q(a) \leftarrow \\ s(b) \leftarrow \end{array} \right\},\ P_3 = \left\{ \begin{array}{l} p(X) \leftarrow \neg q(X), r(X) \\ p(X) \leftarrow \neg t(X), q(X) \end{array} \right\}$$

All stratification refinements form valid stratifications.

For instance, $P = P_1 \sqcup P_2 \sqcup P_3 \sqcup P_4$, where: $P_1 = \left\{ \begin{array}{l} s(b) \leftarrow \\ t(a) \leftarrow \end{array} \right\}$, $P_2 = \{r(X) \leftarrow t(X)\}$,

$P_3 = \left\{ \begin{array}{l} q(X) \leftarrow s(X), \neg t(X) \\ q(a) \leftarrow \end{array} \right\}$ *and* $P_4 = \left\{ \begin{array}{l} p(X) \leftarrow \neg q(X), r(X) \\ p(X) \leftarrow \neg t(X), q(X) \end{array} \right\}$

6.2.4. Iterated Fixpoint Models

The stratified semantics of a (stratifiable) Datalog program with negation is the iterated fixpoint model obtained through the step-wise, bottom-up computation of the least fixpoint model corresponding to each program slice.

Definition 6.2.13 (Stratified Semantics)**.** *Let P be a program stratified by*

$$P = P_1 \sqcup \ldots \sqcup P_n.$$

We iteratively build a model for P, as follows:

$$M_1 = \tilde{T}_{P_1} \uparrow \omega(\emptyset)$$
$$M_n = \tilde{T}_{P_n} \uparrow \omega(M_n - 1)$$

The intended semantics *of P is M_n, denoted as $P^{strat}(I)$.*

Remark 6.2.14. *Note that M_n is* independent from the choice of stratification. *This result is proved in [3].*

We illustrate the computation of the iterative least fixpoint model in the example below.

Example 6.2.15. *Let us compute the semantics of P from Example 6.2.12, according its first stratification, i.e $P = P_1 \sqcup P_2 \sqcup P_3$, where:*

$$P_1 = \left\{ \begin{array}{l} r(X) \leftarrow t(X) \\ s(b) \leftarrow \\ t(a) \leftarrow \end{array} \right\} \quad P_2 = \left\{ \begin{array}{l} q(X) \leftarrow s(X), \neg t(X) \\ q(a) \leftarrow \end{array} \right\} \quad P_3 = \left\{ \begin{array}{l} p(X) \leftarrow \neg q(X), r(X) \\ p(X) \leftarrow \neg t(X), q(X) \end{array} \right\}$$

- $M_1 = T_{P_1} \uparrow \omega(\emptyset) = \{r(a), s(b), t(a)\}$
- $M_2 = T_{P_2} \uparrow \omega(M_1) = M_1 \cup \{q(a), q(b)\}$
- $M_3 = T_{P_3} \uparrow \omega(M_2) = M_2 \cup \{p(b)\}$

Hence, $M_P = M_3 = \{r(a), s(b), t(a), q(a), q(b), p(b)\}$.

Remark 6.2.16. *As established in [3], the computation of the iterated fixpoint model is polynomial in complexity.*

Remark 6.2.17. *The stratified semantics for Datalog programs with negation generalizes that for* semipositive *Datalog programs. As such, an equivalent alternative to Definition 6.2.13 is that found in [1][3]:*

$$M_1 = I, \text{ where } I \text{ is an instance over } P_1$$
$$M_n = M_{n-1} \cup T_{P_{n-1}} \uparrow \omega(M_{n-1}|_{edb(P_i)})$$

Note the use of the immediate consequence operator from Chapter 5. Also note that this is the definition upon which we base our formalization in Chapter 9, conveniently reusing results from our formalization of standard Datalog in Chapter 8.

[3] $M_{n-1}|_{edb(P_i)}$ corresponds to the notion of "interpretation slicing" introduced in Chapter 9

The main property of the stratified semantics is captured by the following result, whose statement is taken from [1]:

Theorem 6.2.18. *For each stratifiable Datalog program P and instance I over $edb(P)$:*

- $P^{strat}(I)$ *is a* minimal model[4] *of* $P*$[5] *and its restriction to* $edb(P)$ *equals* I
- $P^{strat}(I)$ *is a* minimal fixpoint *of* T_P *and its restriction to* $edb(P)$ *equals* I

where $P^{strat}(I)$ denotes the corresponding stratified semantics of P.

Proof. The proof follows by induction and is presented in Chapter 15.2 of [1]. We detail the corresponding Coq proof in Section 9.6.1. □

6.3. Alternative Semantics

As outlined in [1], numerous semantics have been proposed as a way of defining an appropriate meaning for Datalog programs with negation. Among these, the perfect model semantics is a refinement of the stratified one we previously presented and, for the more general case of unstratified programs, the stable model and well-founded semantics are standard approaches.

6.3.1. Perfect Model Semantics

The *perfect model semantics*, introduced in [71], operates on locally-stratified programs that are a generalization of stratified programs. Rather than relying on stratifications of predicate symbols and their induced program partitions, it is based on stratifications of ground atoms and their induced Herbrand base partitions. As such, program slices correspond to rules whose grounded heads are in a given Herbrand base stratum. The computation of the perfect model follows Definition 6.2.13. Stratified programs are also locally stratified and their iterated fixpoint model coincides with the perfect model.

6.3.2. Stable Model Semantics

The *stable model semantics*, introduced in [36], is based on the transformation of a program with negation into a negation-free form. First, the initial program is grounded; then, relative to a given interpretation I, the program is "reduced" by removing all rules containing the negation of atoms belonging to I and, subsequently, removing all negated atoms from the remaining rules. If the unique minimal model of the "reduced" program coincides with I, then I - called a *stable model* - is the unique minimal model of the original program. For stratifiable programs, the stable model coincides with the iterated fixpoint model.

[4] among models M of P such that $M|_{edb(P)} = I$
[5] the first-order logic translation of P according to Section 5.2.1

6.3.3. Well-founded Model Semantics

The *well-founded model semantics*, introduced in [82], extends the stable model semantics to a three-valued logic setting (see [72]). Given a partial interpretation I, a Hebrand base subset is "unfounded", if, for all rule groundings with heads in the subset, it holds that either a positive body atom belongs to the subset or a body literal is false, according to I. Based on this, logical consequence is defined as the union of the immediate consequence with the negation of elements in the greatest unfounded set. The well-founded model is the fixpoint obtained by iterating this monotonous logical consequence operator. For locally stratified programs, the well-founded and perfect model semantics coincide. For Datalog programs with unstratified negation, the (total) well-founded model coincides with the unique stable one.

Part III.

Formalization Overview

7. Formalization of the Relational Model

In this chapter, we present a Coq formalization of the relational data model, which underlies relational database systems. More precisely, we formalize the data definition part of the model, including integrity constraints. We model two different query language formalisms: relational algebra and conjunctive queries. Also, we mechanize logical query optimization and prove the main "database theorems", namely algebraic equivalence, the homomorphism theorem and conjunctive query minimization.

7.1. Data Representation

As introduced in Section 4.1, in the relational model, data is represented by relations. These consists of rows, whose components are labelled with attribute names and contain values from given domains. The structure of such relations (their schema) is dissociated from their content (their instance), as outlined in our development as well.

In this section we outline the way in which we formalized all these base ingredients in Coq. We start by presenting the encoding of attributes, domains and values, in Section 7.1.1. Then, we move on to the modelization of tuples, in Section 7.1.2, and to that of relations, schemas and instances, in Section 7.1.3.

7.1.1. Attributes, Domains, Values

Quoting [1], a set *attribute*, containing attribute names is fixed and equipped with a total order $\leq_{\mathbf{att}}$. When different attributes should have distinct domains (or types), a mapping, *dom*, from *attribute* to *domain*, is assumed. Furthermore, an infinite set *value* is fixed. Usually, the set of attributes is assumed to be countably infinite, but, in our formal development, this assumption was not needed. We also assume several distinct domain names (*e.g.*, "string", "integer"), which belong to the *domain* set. In the database context, *domain* corresponds to the Coq notion of a type. In order to have a decidable equality, we rather used our own `type : Type`. In our setting, *dom* is called `type_of_attribute`. Each value has a formal type (obtained by the function `type_of_value`), inhabiting `type`. All these assumptions are gathered in a Coq record `Tuple.Rcd`, presented below, whose contents will be iteratively enriched throughout this section.

```
Module Tuple.
Record Rcd : Type := mk_R {
(* primitives: attributes, types, values *)
```

```
  attribute : Type;
  type : Type;
  value : Type;
(* typing attributes and values *)
  type_of_attribute : attribute → type;
  type_of_value : value → type;
(* default values.*)
  default_value : type → value;
  ... }.
End Tuple.
```

We illustrate these definitions with our running movie example. Recall that our purpose is *not* to store an actual database schema or instance in Coq. The following example is rather intended to be a *proof of concept*.

```
Inductive attribute :=
  | Title | Director | Actor | Theater | Address | Phone | Schedule.

Inductive type := type_string | type_nat | type_Z.

Inductive value :=
  | Value_string : string → value
  | Value_nat : nat → value
  | Value_Z : Z → value.

Definition type_of_attribute x :=
  match x with
    | Title   | Director | Actor
    | Theater | Address  | Phone ⇒ type_string
    | Schedule ⇒ type_nat
  end.

Definition type_of_value v :=
  match v with
    | Value_string _  ⇒ type_string
    | Value_nat _  ⇒ type_nat
    | Value_Z _  ⇒ type_Z
  end.
```

There is also a more generic modeling for attributes, and in that case, for the sake of readability, we could use the Coq notations shown in [13].

7.1.2. Tuples

In the named perspective, tuples are characterized by their relevant attributes, *e.g*, for the `Movies` relation, these are `{Title, Director, Actor}`. We call this the `support` of the tuple. Following textbooks, we naturally model it using finite sets. To this end, we adapted Letouzey's MSet library [55]. In order to be as modular as possible, we still dissociate the *specification* of finite sets from the *implementation*. The specification is given by a record `Fset.Rcd`, parametrized by the type of elements, and contains a comparison function `elt_compare`.

From now on, we will denote set equivalence with $\overset{set}{=}$. This corresponds to extensional set equality: $\forall s \forall s',\ s \overset{set}{=} s' \Leftrightarrow (\forall e,\ e \in s \Leftrightarrow e \in s')$. For the sake of readability, the usual sets operators will be denoted by their standard mathematical notations ($\cap$, $\cup$, $\setminus$, $\in$,...). Extending the record `Tuple.Rcd` above, we further assume:

```
Module Tuple.
Record Rcd : Type := mk_R {
  (* primitives : attributes, domains and values *)
  ...
  (* finite sets of attributes *)
  A : Fset.Rcd attribute;
  (* tuples, their support and value extraction functions *)
  tuple : Type;
  support : tuple → set A;
  dot : tuple → attribute → value;
  (* building tuples and corresponding soundness conditions *)
  mk_tuple : set A → (attribute → value) → tuple;
  support_mk_tuple_ok : ∀ V f, support (mk_tuple V f) =set= V;
  dot_mk_tuple_ok : ∀ a V f, a ∈ V → dot (mk_tuple V f) a = f a;
  (* finite sets of tuples *)
  FTuple : Fset.Rcd tuple;
  (* tuple equivalence *)
  tuple_eq_ok : ∀ t1 t2 : tuple,
    (Fset.elt_compare FTuple t1 t2 = Eq) ⟷
    (support t1 =set= support t2 ∧
     ∀ a, a ∈ (support t1) → dot t1 a = dot t2 a)
}
End Tuple.
```

where `A` models finite sets of `attribute`s. We still keep the type of tuples abstract and assume the existence of two functions: `support`, returning the relevant tuple attributes, and `dot`, the associated field extraction. These functions allow us to characterize tuple equivalence (`tuple_eq_ok`), since a tuple `t` behaves as the pair `(support t, dot t)`. Further, we assume the existence of the `mk_tuple` function, which builds tuples. This

and the previous modeling of `attribute` induce a notion of tuple *well-typedness*

A tuple `t` is *well-typed* if and only if, for any attribute `a` in its support, the type of the extracted value `t.a` corresponds to the type of the attribute `a`:

```
Definition well_typed_tuple (t : tuple) :=
  ∀ a, a ∈ (support t) →
  type_of_value (dot t a) = type_of_attribute a.
```

However and surprisingly, such a notion was useless in proving standard textbook results. This is an *a posteriori* justification of the *relevance* of the assumption that it suffices to use a *unique domain* for values. The previously presented record `Tuple.Rcd` captures exactly the abstract behavior of tuples, *i.e.*, the needed properties for proving all the theorems presented hereafter. To illustrate the generality and flexibility of our specification, we give, in [13], different possible implementations for tuples. All of them satisfy the required properties and are orthogonal to the implementation of attributes. For example, one can implement tuples as pairs containing a set of attributes and a function or as association lists, between attributes and values.

7.1.3. Relations, Schemas and Instances

A distinction is made between the database *schema*, which specifies the structure of the database, and the database *instance*, which specifies its actual content: sets of tuples. In textbooks, each table is called a *relation* and has a *name*. Hence, we assume a set `relname` of relation names, equipped with a comparison function, specified by `ORN`. The latter is suitable for computing equality checks and building finite set records. The structure of a table is given by a relation name and a *finite* set of attributes: its *sort*. The relation name, together with its sort, is called the *relation schema*. A *database schema* is a non-empty finite set of relation schemas. We choose to model database schemas with a function `basesort`, which associates to each `relname` its sort. We adopted this representation, as it is the most *abstract* and makes no further choice on the *concrete implementation* of the `basesort` function, *e.g*, be it through association lists, finite maps or even functions.

```
Module DatabaseSchema.
Record Rcd attribute (A : Fset.Rcd attribute) : Type := mk_R {
 (* relation names *)
 relname : Type;
 (* ordering on relation names *)
 ORN : Oset.Rcd relname;
 (* relation sort *)
 basesort : relname → set A
 }.
```

```
End DatabaseSchema.
```

More precisely, the `basesort` function will be used to relate the `support` of tuples - in the instance they belong to - and the structure of the corresponding relation name.

```
Definition well_sorted_instance (I : relname → setT) :=
  ∀(r : relname)(t : tuple), t ∈ (I r) → support t =^set basesort r.
```

It is important to mention that, in all our further development, the notion of *well-sorted instance* resulted *central to the correctness* of many theorems.

7.2. Data Extraction: Query Languages

Queries allow for the extraction of information from tables. The result of a query is also a table or a collection of tables. Information extraction is usually performed by a query language, the standard being SQL or QBE. All these languages rely on a more formal basis: *relational algebra* or *first-order logic*. Both formalisms are based on the notion of *tuples*. Thus, we assume the existence of a record `T` of type `Tuple.Rcd`, for representing tuples, as well as of a record `DBS` of type `DatabaseSchema.Rcd`, for representing base relations. Moreover, we assume that `T` and `DBS` use the same representation, `A`, for finite sets of attributes. This is achieved parameterizing `DBS` by `(A T)`. For the sake of readability, we omit all extra (implicit) record arguments and denote by `setA` and `setT` the finite sets of attributes and, respectively, of tuples.

7.2.1. Relational Queries

Relational algebra consists of a set of (algebraic) operators with relations as operands. The algebra we shall consider in this article is the *SPJRU(ID)*, where S stands for selection, P for projection, J for natural join, R for renaming and last U for union. Though intersection (I) and difference (D) are not part of the SPJRU minimal algebra, we decided to include them at this point, as they are usually part of commercial query languages. In the context of the named version, relations are combined using the natural join, whereas, in the unnamed one, the Cartesian product is used. The complete definition of queries is given in Figure 7.1. In our development, we chose, as far as possible, *not to embed proofs* in types. Hence, types are much more concise and readable.

Syntax

Base relations are queries. Concerning the **selection operator**, in textbooks, as seen in Section 4.2.1, it has the form $\sigma_{A=a}$ or $\sigma_{A=B}$, where $A, B \in$ *attribute* and $a \in$ *value*. The notation $A = a$, respectively $A = B$, is improper and corresponds to $x.A = a$, respectively $x.A = x.B$, where x is a *free variable*. Given a set of tuples $\mathcal{I}$, with the same support S, we call S the *sort* of $\mathcal{I}$. The selection with respect to $A = B$ applies

```
(* attribute renaming; no assumptions are made at this point,
   neither about type compatibility, nor about injectivity. *)
Definition renaming := attribute T → attribute T.

Inductive query : Type :=
  | Query_Basename : relname → query
  | Query_Sigma : formula → query → query
  | Query_Pi : setA → query → query
  | Query_NaturalJoin : query → query → query
  | Query_Rename : renaming → query → query
  | Query_Union : query → query → query
  | Query_Inter : query → query → query
  | Query_Diff : query → query → query
with variable : Type :=
  | Var : query → varname → variable
with term : Type :=
  | Term_Constant : value → term
  | Term_Dot : variable → attribute  → term
with atom : Type :=
  | Atom_Eq : term → term → atom
  | Atom_Le : term → term → atom
with formula : Type :=
  | Formula_Atom : atom → formula
  | Formula_And : formula → formula → formula
  | Formula_Or : formula → formula → formula
  | Formula_Not : formula → formula
  | Formula_Forall : variable → formula → formula
  | Formula_Exists : variable → formula → formula.
```

Figure 7.1.: Queries

to any set of tuples $\mathcal{I}$ of sort S, with $A, B \in S$, and yields an output of sort S. The semantics of the operator is:

$$\sigma_f(\mathcal{I}) = \{t \mid t \in \mathcal{I} \wedge f\{x \to t\}\}$$

where $f\{x \to t\}$ denotes "t satisfies formula f" and x is the only free variable in f. Formula satisfaction is based on the standard underlying interpretation (see Section 3). Since, in the context of database program verification, general first-order formulas are used, we also chose these to model selection (filtering) conditions, rather than restricting ourselves to the simpler case found in textbooks. We first introduce variable names :

```
Inductive varname : Set := VarN : N →  varname.
```

Next, formulas are built in the standard way, from equality and inequality atoms, which either compare constants or tuple field extractions. However, one should notice that variables are used to denote tuples in the output of specific queries, therefore containing information about the query itself. For example, the variable `x` below is intended to represent any tuple in the `Movies` relation, while the formula `f` corresponds to:

$$\texttt{x} \in \texttt{Movies} \Rightarrow \texttt{x.Director} = \texttt{"Fellini"}.$$

```
(* x ∈ Movies *)
Notation x := (Var (Query_Basename Movies) (VarN 0)).
(* x.Director = "Fellini" *)
Definition f := (Formula_Atom
                  (Atom_Eq (Term_Dot x Director)
                           (Term_Constant (Coq_string "Fellini")))).
```

The **projection operator** has the form: $\pi_{\{A_1,\ldots,A_n\}}$, $n \geq 0$ and operates on all inputs, $\mathcal{I}$, whose sort contains the subset of attributes $W = \{A_1, \ldots, A_n\}$ and produces an output of sort W. The semantics of projection is: $\pi_W(\mathcal{I}) = \{t|_W \mid t \in \mathcal{I}\}$, where the notation $t|_W$ represents the tuple obtained from t, by only keeping the attributes in W. Note that `setA`, the type of W, denotes finite sets of attributes and embeds, as an implicit argument, `(A T)`, the record representing all types and operations on finite sets. Depending on the actual implementation of sets, this definition may contain some proofs in the `setA` datatype. For instance, the proof that a set is an AVL tree may be part of the type.

The **natural join operator**, denoted $\bowtie$, takes arbitrary inputs $\mathcal{I}_1$ and $\mathcal{I}_2$, of respective sorts V and W, and produces an output of sort $V \cup W$. The semantics is: $\mathcal{I}_1 \bowtie \mathcal{I}_2 = \{t \mid \exists v \in \mathcal{I}_1, \exists w \in \mathcal{I}_2, t|_V = v \wedge t|_W = w\}$. When $sort(\mathcal{I}_1) = sort(\mathcal{I}_2)$, then $\mathcal{I}_1 \bowtie \mathcal{I}_2 = \mathcal{I}_1 \cap \mathcal{I}_2$, and, when $sort(\mathcal{I}_1) \cap sort(\mathcal{I}_2) = \emptyset$, then $\mathcal{I}_1 \bowtie \mathcal{I}_2$ is the cross-product of $\mathcal{I}_1$ and $\mathcal{I}_2$. The join operator is associative and commutative.

An **attribute renaming** for a finite set V of attributes is a one-to-one mapping from V to *attribute*. In textbooks, an attribute renaming g for V is specified by the set of pairs

$(a, g(a))$, where $g(a) \neq a$; this is usually written as $a_1 a_2 \dots a_n \to b_1 b_2 \dots b_n$, to indicate that $g(a_i) = b_i$, for each $i \in [1, n]$, $n \geq 0$. A renaming operator for inputs over V is an expression ρ_g, where g is an attribute renaming for V; this maps to outputs over $g[V]$. Specifically, for $\mathcal{I}$ over V, $\rho_g(\mathcal{I}) = \{v \mid \exists u \in \mathcal{I}, \forall a \in V, v(g(a)) = u(a)\}$.

We made a different, more abstract, choice when modeling this operator. To avoid proofs in types, we made no assumptions on the "renaming" function, except for its type `attribute → attribute`, in the inductive definition. However, the one-to-one assumption will explicitly appear as an hypothesis for some theorems. Set operators can be applied over sets of tuples, $\mathcal{I}_1$ and $\mathcal{I}_2$, with the same sort. As it is standard in mathematics, $\mathcal{I}_1 \cup \mathcal{I}_2$, respectively $\mathcal{I}_1 \cap \mathcal{I}_2$ and $\mathcal{I}_1 \setminus \mathcal{I}_2$, is the set having this same sort and containing the union, respectively the intersection and difference, of the two sets of tuples. Sort compatibility constraints are absent from our modeling, so as to avoid proofs, and will be enforced in the semantics part.

Semantics

We present our Coq modeling of query evaluation. To this end, we have to explicitly describe constraints about sorts, which were, deliberately, left out of the query syntax. For base queries, the sort corresponds to the `basesort` of the relation name; for selections, the sort is left unchanged and, for joins, the sort is, as expected, the union of sorts. The cases which are of interest are projection, renaming and set theoretic operators.

For projections, rather than imposing that the set `W` of attributes on which we project, be a subset of the `sort` of `q1`, we chose to define the sort of `Query_Pi W q1` as their intersection (`W ∩ sort q1`). For renaming, we check that the corresponding function `rho` behaves as expected, *i.e.*, that it is a one-to-one mapping over attributes in `q1`; otherwise the sort of the query is empty. Lastly, for set theoretic operators, if the input sorts are not compatible, the sort of the query is empty. This is formally defined by:

```
Fixpoint sort (q : query) : setA := match q with
  | Query_Basename r ⇒ basesort r
  | Query_Sigma _ q1 ⇒ sort q1
  | Query_Pi W q1 ⇒ W ∩ sort q1
  | Query_Join q1 q2 ⇒ sort q1 ∪ sort q2
  | Query_Rename rho q1 ⇒
     let sort_q1 := sort q1 in
       if one_to_one_renaming_bool sort_q1 rho
       then fset_map A A rho sort_q1
       else ∅
  | Query_Union q1 q2 | Query_Inter q1 q2
  | Query_Diff q1 q2 ⇒
     let sort_q1 := sort q1 in
       if sort_q1 =?^{set} sort q2
```

```
        then sort_q1
        else ∅
  end.
```

At this point we are ready to interpret queries. We first assume an interpretation for base relations. When proving the usual structural equivalence theorems (Section 4.2.1) for query optimization, we impose that instances be *well-sorted*. This means that all tuples in an instance or query evaluation must have the *same* support, which is the sort of the query. This property is inherited from base instances, as stated below:

```
Lemma well_sorted_query :
  ∀ (I : relname → setT), well_sorted_instance I →
  ∀ (q : query) (t : tuple),
  t ∈ (eval_query I q) → support t =^set sort q.
```

Query evaluation is inductively defined from a given interpretation `I` for base relations. We sketch its structure (the complete definition of `eval_query` is given in [13]), in order to emphasize the fact that the same tests as for sorts are performed. For example, for renaming, if the corresponding function is not suitable, the query evaluates to the empty set of tuples.

```
Fixpoint eval_query I (q : query) : setT := match q with
  | Query_Basename r ⇒ I r
  | Query_Sigma f q1 ⇒  ...
  | Query_Pi W q1 ⇒ ...
  | Query_Join q1 q2 ⇒ ...
  | Query_Rename rho q1 ⇒
     let sort_q1 := sort q1 in
       if one_to_one_renaming_bool sort_q1 rho
       then ...
       else ∅
  | Query_Union q1 q2 ⇒
      if sort q1 =^set_? sort q2
      then (eval_query I q1) ∪ (eval_query I q2)
      else ∅
  | Query_Inter q1 q2 ⇒ if sort q1 =^set_? sort q2 ...
  | Query_Diff q1 q2 ⇒ if sort q1 =^set_? sort q2 ...
end.
```

Our definition enjoys the standard properties stated in all database textbooks, which are expressed in our framework by the following lemmas. We only present some of them; the full list, as well as the complete code, is given in [13]. In particular, we detail the way in which terms, atoms and formulas are interpreted. For the sake of readability, we

used some syntactic sugar, such as $\overset{I}{=}$, $\in_I$, as well as `f {x → t}`, for the interpretation of a formula `f` under an assignment `x → t`.

```
Notation query_eq q1 q2 := (eval_query I q1 =set= eval_query I q2).
Infix "=I=" := query_eq.
Notation "t '∈I' q" := t ∈ (eval_query I q).

Lemma mem_Basename :
  ∀ I r t, t ∈I (Query_Basename r) ⟷ t ∈ (I r).

Lemma mem_Inter : ∀ I q1 q2, sort q1 =set= sort q2 →
  ∀ t, t ∈I (Query_Inter q1 q2) ⟷ (t ∈I q1 ⋀ t ∈I q2).

Lemma mem_Sigma : ∀ I, well_sorted_instance I →
  ∀ f x q t, set_of_attributes_f f ⊆ sort q →
  Fset.elements FV (free_variables_f f) = x :: nil →
  (t ∈I (Query_Sigma f q) ⟷ (t ∈I q ⋀ f {x → t} = true)).

Lemma mem_Pi : ∀ I, well_sorted_instance I →
  ∀ W q t, t ∈I Query_Pi W q ⟷
  ∃ t', (t' ∈I q ⋀ t =t= mk_tuple (W ∩ sort q) (dot t')).

Lemma mem_Join : ∀ I, well_sorted_instance I →
  ∀ q1 q2 t, t ∈I Query_Join q1 q2 ⟷
  ∃ t1, ∃ t2, (t1 ∈I q1 ⋀ t2 ∈I q2
  ⋀ (∀ a, a ∈ sort q1 ∩ sort q2 → dot t1 a = dot t2 a)
  ⋀ t =t= mk_tuple (sort q1 ∪ sort q2)
        (fun a ⇒ if a ∈? (sort q1) then dot t1 a else dot t2 a)).

Definition rename_tuple (rho : renaming) (t : tuple T) : tuple T :=
  let V := support T t in
  mk_tuple T
    (fset_map _ (A T) rho V)
    (fun a ⇒ dot T t (inv_fun (A T) (A T) a V rho a)).

Lemma mem_Rename : ∀ I, well_sorted_instance I →
  ∀ rho q, one_to_one_renaming (sort q) rho →
  ∀ t, t ∈I (Query_Rename rho q) ⟷
  (∃ t', t' ∈I q ⋀ t =t= rename_tuple rho t').

Lemma NaturalJoin_Inter : ∀ I, well_sorted_instance I →
  ∀ q1 q2, sort q1 =set= sort q2 →
```

```
Query_NaturalJoin q1 q2 ≐ Query_Inter q1 q2.
```

The lemmas above highlight the heterogeneous nature of relational operators. In order to prove that these enjoy their usual semantics, only sort compatibility conditions were needed for the purely set theoretic ones, while, for the database ones, the well-sortedness conditions sufficed. Interestingly, the `NaturalJoin_Inter` lemma, which bridges both worlds, needed both conditions.

Optimizing Relational Algebra Queries Query optimization exploits algebraic equivalences. Such equivalences are found in all textbooks and in particular in [73]. We list the most classical ones hereafter.

$$\sigma_{f_1 \wedge f_2}(q) \equiv \sigma_{f_1}(\sigma_{f_2}(q)) \quad (7.1) \qquad \pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q) \quad \text{if } W_1 \subseteq W_2 \quad (7.5)$$

$$\sigma_{f_1}(\sigma_{f_2}(q)) \equiv \sigma_{f_2}(\sigma_{f_1}(q)) \quad (7.2) \qquad \pi_W(\sigma_f(q)) \equiv \sigma_f(\pi_W(q)) \quad \text{if } \mathcal{A}tt(f) \subseteq W \quad (7.6)$$

$$(q_1 \bowtie q_2) \bowtie q_3 \equiv q_1 \bowtie (q_2 \bowtie q_3) \quad (7.3) \qquad \sigma_f(q_1 \bowtie q_2) \equiv \sigma_f(q_1) \bowtie q_2 \quad \text{if } \mathcal{A}tt(f) \subseteq sort(q_1) \quad (7.7)$$

$$q_1 \bowtie q_2 \equiv q_2 \bowtie q_1 \quad (7.4) \qquad \sigma_f(q_1 \nabla q_2) \equiv \sigma_f(q_1) \nabla \sigma_f(q_2) \quad \text{where } \nabla \text{ is } \cup, \cap \text{ or } \setminus \quad (7.8)$$

All these have been formally proved and their formal statements are given in [13]. Although not technically involved, all the proofs relied on the assumption that instances are *well-sorted*. To illustrate this, we give the formal statement of (7.7).

```
Lemma Sigma_NaturalJoin_comm : ∀ I, well_sorted_instance I →
  ∀ f q1 q2, set_of_attributes_f f ⊆ sort q1 →
  Query_Sigma f (Query_NaturalJoin q1 q2) ≐
  Query_NaturalJoin (Query_Sigma f q1) q2.
```

7.2.2. Conjunctive Queries

In this context, the query language is slightly different. Rather than relying on algebraic operators, queries are expressed by logical formulas of the form $\{(a_1, \ldots, a_n) \mid \exists b_1, \ldots \exists b_m, P_1 \wedge \ldots \wedge P_k\}$, where the a_i, b_i denote variables which will be interpreted by values and where P_i's denote either equalities or membership to a base relation. Recall from Section 4.2.2 that, for example, the query: "Which of "Fellini" 's movies are played at the cinema "Action Christine" ?" expressed in relational algebra as:

$$\pi_{\{\text{Title, Director, Actor}\}}(\sigma_{\substack{\text{x.Director="Fellini"} \wedge \\ \text{x.Theater = "Action Christine"}}}(\texttt{Movies} \bowtie \texttt{Pariscope}))$$

corresponds to the conjunctive query:

$$\left\{(t,d,a) \mid \begin{array}{l} \exists th, \exists t', \exists s, \texttt{Movies}(t,d,a) \wedge \texttt{Pariscope}(th,t',s) \\ \wedge t = t' \wedge d = \text{"Fellini"} \wedge th = \text{"Action Christine"} \end{array}\right\}$$

Quoting [1], "if we blur the difference between a variable and a constant, the body of a conjunctive query can be seen as an instance with additional constraints". This leads to the notion of extended tuples mapping attributes to either *constants* or *variables*. Hence, a *tableau* over a schema is defined exactly as was the notion of an instance over this schema, except that it contains extended tuples. A *conjunctive query* is simply a pair *(T, s)* where T is a tableau and s, an extended tuple called the *summary* of the query. Variables occurring in s are called *distinguished variables* or *distinguished symbols* in textbooks. The summary s in query *(T, s)* represents the answer to the query which consists of all tuples for which the pattern described by T is found in the database. This formulation of queries is closest to the QBE visual form. Equality conditions are embedded in the tableau itself as shown by the following example:

Title	Director	Actor	Theater	Schedule	
t	"Fellini"	a			Movies
t			"Action Christine"	s	Pariscope
t	d	a			summary

Syntax

The formal way to "blur" the differences between variables and constants (`values` in our modelling) is achieved by embedding them in a single Coq type, *i.e*, `tvar`.

```
Inductive tvar : Type :=
 | Tvar : nat → tvar
 | Tval : value → tvar.

Inductive trow : Type :=
 | Trow : relname → (attribute → tvar) → trow.
```

Notice that a row, modeled by the type `trow`, is tagged by a relation name (its first argument) and gathers variables and constants, thanks to its second argument. For instance, the first row of the above query is:

```
Trow Movies
  (fun a : attribute ⇒
     match a with
       | Title    ⇒ Tvar 0
       | Director ⇒ Tval ‘‘Fellini’’
       | Actor    ⇒ Tvar 2
     end)
```

A tableau is a `trow` set. This is built using a comparison function, similar to the one for tuples. Next, a summary is tagged by a set of relevant attributes and maps from `attribute` to `tvar`. Lastly, a conjunctive query (`tableau_query`) consists of a tableau

and a summary.

```
Notation setR := (Fset.set (Ftrow T DBS)).

Definition tableau := setR.

Inductive summary : Type :=
  | Summary : setA → (attribute → tvar) → summary.

Definition tableau_query := (tableau * summary)
```

Semantics

We illustrate the semantics of conjuctive queries through our previous example. Its query is expressed by the `summary`:

```
Summary (mk_set A (Title :: Director :: Actor :: nil))
  (fun a : attribute ⇒
     match a with
       | Title    ⇒ Tvar 0
       | Director ⇒ Tvar 1
       | Actor    ⇒ Tvar 2
     end)
```

and its result consists of the set of movies:

```
mk_set A
  ((mk_movie "Casanova" "Fellini" "Donald Sutherland") ::
   (mk_movie "La strada" "Fellini" "Giulietta Masini") :: nil)
```

This set is computed by composing the `summary` function with some mappings from variables in the tableau rows to values, hence mapping summaries to tuples. Thus, we first need to define the notion of `valuation`, which, as usual, maps variables to values. More precisely, in our case, as we embedded variables and constants in a single abstract type `tvar`, and as variables are characterized by their `nat` identifier, the type of `valuation` is `nat → value`. Indeed, applying a valuation (thanks to `apply_valuation`) on constants, consists of applying the identity function.

```
Definition valuation := nat → value.

Definition apply_valuation (ν : valuation) (x : tvar) : value :=
```

```
  match x with
    | Tvar n ⇒ ν n
    | Tval c ⇒ c
  end.
Notation "ν '⟦' x '⟧'" := (apply_valuation ν x).
```

Valuations naturally extend to `trow`'s and `summary`'s, yielding tuples, and to tableaux, yielding sets of tuples.

```
Definition apply_valuation_t (ν : valuation)(x : trow) : tuple :=
  match x with
    | Trow r f ⇒ mk_tuple (basesort r) (fun a ⇒ ν ⟦f a⟧)
  end.
Notation "ν '⟦' x '⟧t'" := (apply_valuation_t ν x).

Definition apply_valuation_s (ν : valuation)(x : summary) : tuple :=
  match x with
    | Summary V f ⇒ mk_tuple V (fun a ⇒ ν ⟦f a⟧)
  end.
Notation "ν '⟦' x '⟧s'" := (apply_valuation_s ν x).
```

Given a query (T, s), its result on an instance $\mathcal{I}$ is given by: $\{t \mid \exists \nu, \nu(T) \subseteq \mathcal{I} \wedge t = \nu(s)\}$, where ν is a valuation. In our development, we characterize this set, using the predicate `is_a_solution I (T, s)`, where $\overset{t}{=}$ denotes tuple equivalence.

```
Inductive is_a_solution (I : relname → setT)
  : tableau_query → tuple → Prop :=
  | Extract : ∀ (ST : tableau) (s : summary) (ν : valuation),
    (∀ (r : relname) (f : attribute → tvar),
        (Trow r f) ∈ ST → ν ⟦Trow r f⟧t ∈I (Query_Basename r)) →
     ∀ (t : tuple), t =t= ν ⟦s⟧s → is_a_solution I (ST, s) t.
```

Optimizing Conjunctive Queries For the algebraic queries that are expressible by a conjunctive query, there exists an exact optimization technique. In this case, query optimization is based on the following consideration: the number of rows in the tableau corresponds to the number of joins (plus one) in the relational expression. Hence, the problem of conjunctive query optimization boils down to that of reducing the former.

The notions of containment, equivalence and minimality for tableaux are analogous to those for queries. Specifically, let Q_1, Q_2 be two conjunctive queries and $(T_1, s_1), (T_2, s_2)$, their tableaux representation. The tableau (T_1, s_1) is said to be contained in (T_2, s_2),

denoted as $(T_1, s_1) \subseteq (T_2, s_2)$, iff (T_1, s_1) and (T_2, s_2) have the same set of attributes and, for all relation instances, solutions of (T_1, s_1) are also solutions of (T_2, s_2). This inclusion induces the equivalence:

$$(T_1, s_1) \equiv (T_2, s_2) \text{ iff } (T_1, s_1) \subseteq (T_2, s_2) \text{ and } (T_2, s_2) \subseteq (T_1, s_1).$$

This is formalized in Coq by:

```
Definition is_contained_instance I Ts1 Ts2 :=
  ∀ (t : tuple), is_a_solution I Ts1 t → is_a_solution I Ts2 t.

Definition is_contained Ts1 Ts2 :=
  ∀ I, is_contained_instance I Ts1 Ts2.

Definition are_equivalent Ts1 Ts2 :=
  is_contained Ts1 Ts2 ⋀ is_contained Ts2 Ts1.
```

These semantic properties can be checked syntactically thanks to the notion of tableau homomorphism, introduced as follows.

Definition 7.2.1 (Tableaux Homomorphism)**.** *Let (T_1, s_1) and (T_2, s_2) be two tableaux. A homomorphism (tableau substitution) $\theta : (T_1, s_1) \to (T_2, s_2)$ is a mapping from variables to variables or constants. This satisfies the conditions: 1) for all rows $\vec{t_i}$ tagged by a relation name R_i in T_1, $\theta(\vec{t_i})$ occurs tagged by R_i in T_2 and 2) $\theta(s_1) = s_2$.*

Based on this, the theorem below characterizes conjunctive query containment:

Theorem 7.2.2 (Homomorphism Theorem)**.** *Let Q_1, Q_2 be two conjunctive queries and $(T_1, s_1), (T_2, s_2)$ be their tableaux representations. Then:*

$$Q_1 \subseteq Q_2 \textit{ iff there exists a homomorphism } \theta : (T_2, s_2) \to (T_1, s_1).$$

Consequentely, given a conjunctive query Q, an equivalent minimal one can be constructed, once the query is transformed into the tableau form (T, s). Indeed, this is realized by iterating the following procedure, until a fixpoint is reached: 1) chosing a row $\vec{t}$ from T and 2) checking whether a homomorphism $\theta : (T, s) \to (T \setminus \vec{t}, s)$ exists. Regarding our Coq formalization of the above, let us first give the definition of tableau substitution in our setting and then formally define the application of a substitution to a variable. This notion extends to `trow`'s and `summary`'s. Then, we give the corresponding encodings for the tableau homomorphism and for the Homomorphism Theorem.

```
Definition substitution := nat → tvar.

Definition apply_subst_tvar (θ : substitution) (x : tvar) :=
  match x with
    | Tvar n ⇒ θ n
    | Tval _ ⇒ x
```

```
  end.
Notation "θ '[' x ']_v'" := (apply_subst_tvar θ x).

Definition tableau_homomorphism (θ : substitution) Ts2 Ts1 :=
  match Ts1, Ts2 with (T1, s1), (T2, s2) ⇒
    (fset_map Ftrow Ftrow (fun t ⇒ θ [t]_t) T2) ⊆ T1
    ⋀ θ [s2]_s ≐ s1
  end.

Theorem Homomorphism_theorem :
   ∀ Ts1 Ts2, (∃ θ, tableau_homomorphism θ Ts2 Ts1) ⟷
    is_contained Ts1 Ts2.
```

We briefly sketch the Coq proof of the Homomorphism Theorem. Interestingly, in textbooks a lot of material is hidden. Namely, the notion of *fresh constants* is central to the proof in order to be able to define a list of such *distinct* fresh constants for each variable present in the query. We assume therefore

```
Hypothesis fresh : (Fset.set Ftvar) → value.

Hypothesis fresh_is_fresh :
  ∀ lval, (Tval (fresh lval)) ∈ lval → False.
```

This implies that domains are *infinite*. Based on fresh constants we define a variable assignment μ from variables to new fresh abstract constants on (T_1, s_1). We then show that μ is a solution of (T_1, s_1) with respect to the interpretation I, which contains exactly $\mu(T_1)$. Thanks to the definition of tableaux containment, μ is a solution of (T_2, s_2) with respect to I. Hence there is an assignment ν which corresponds to a solution of (T_2, s_2), $\nu(s_2) = \mu(s_1) \wedge (\forall t_2\ r, t_2 : r \in T_2 \Rightarrow \nu(t_2) \in I(r))$, that is $\nu(s_2) = \mu(s_1) \quad \wedge \quad (\forall\ t_2\ r, \quad t_2 : r \in T_2 \Rightarrow \exists\ t_1, \quad t_1 : r \in T_1 \wedge \nu(t_2) = \mu(t_1))$. By construction μ admits an inverse function defined over the variables of (T_1, s_1). What remains to show is that $x \mapsto \mu^{-1}(\nu(x))$ is an homomorphism from (T_2, s_2) to (T_1, s_1). The main difficulties encountered in Coq were to properly define the notion of query solution, to build the variable assignment μ as a function from the `fresh` function and to prove that μ is injective.

At this point, based on the Homomorphism Theorem, given a conjunctive query, we explicitly construct an equivalent minimal one. As mentioned above, the optimization process consists in inspecting all equivalent sub-tableaux and, among those, keeping a minimal one.

```
Definition min_tableau Ts Ms :=
  are_equivalent Ts Ms
```

```
    ⋀ (∀ Ts', are_equivalent Ts Ts' → cardinal (fst Ms) ≤ cardinal
      (fst Ts')).

Lemma tableaux_optimisation :
  ∀ T s, {T' | min_tableau (T, s) (T', s)}.
```

More precisely, the cornerstone of the algorithm is finding an homomorphism from the initial tableau to a given sub-tableau. To do so we used a function `abstract_matching`. All further details are given in [13]. Not only do we *prove* this result but we also provide a *certified algorithm* to build this minimal tableau both in Coq and by *extraction* from `tableaux_optimization` in OCaml.

7.2.3. From Algebraic to Conjunctive Queries

The two formalisms presented are not exactly equivalent, except for the case where relational queries are only built with selections, projections and joins. In this case, there is an *apparently straightforward* way to construct the corresponding conjunctive query. In Figure 7.2, we give the verbatim account of the algorithm found in [80].

Given an SPJ algebraic expression, a conjunctive query equivalent to this expression is inductively constructed using the following rules. The base case consists in a relation $r(A_1, \ldots, A_n)$ the corresponding tableau consists in a single row and summary which are exactly the same with one variable for each A_i. Assume that we have an expression of the form $\pi_W(E)$ and that we have constructed (T, s) for E, then to reflect the projection, all the distinguished variables that are not in W are deleted from s. For selections $\sigma_f(E)$ where f is either of the form $A = B$ or $A = c$, in the former case, the distinguished symbols for columns A and B in the summary and the tableau are identified, in the latter, the distinguished variable for A is replaced by c. For joins $E_1 \bowtie E_2$, it is assumed without loss of generality that if both (T_1, s_1) and (T_2, s_2) have distinguished symbols in the summary column for attribute A then those symbols are the same, but that otherwise (T_1, s_1) and (T_2, s_2) have no symbols in common. Then the tableau for $E_1 \bowtie E_2$ has a summary in which a column has a distinguished symbol a if a appears as a distinguished symbol in that column of s_1 or s_2 or both. The new tableau has as rows all the rows of T_1 and T_2.

Figure 7.2.: Ullman's algorithm for translating algebraic queries into conjuctive ones

If we apply this algorithm on the relational expression $\sigma_{A=B}(r) \bowtie \sigma_{B=C}(r)$, we obtain, for E_1 and E_2, the tableaux: $\begin{array}{cccc} x_1 & x_1 & x_2 & \mathrm{r} \\ \hline x_1 & x_1 & x_2 & \end{array}$ and $\begin{array}{cccc} y_1 & y_2 & y_2 & \mathrm{r} \\ \hline y_1 & y_2 & y_2 & \end{array}$.
Given these, whatever renaming we choose to apply to the second tableau, as stated in [80], there is no way to be in the situation described by the algorithm, *i.e., if both* (T_1, s_1) *and* (T_2, s_2) *have distinguished symbols in the summary column for attributes*

in A, then those symbols are the same. We fixed this source of incompleteness, using *unification*, instead of renaming. Indeed, if we unify the two summaries of our example, we obtain: $x_2 \mapsto x_1$; $y_1 \mapsto x_1$; $y_2 \mapsto x_1$, yielding the tableau $\frac{x_1 \quad x_1 \quad x_1 \quad \mathrm{r}}{x_1 \quad x_1 \quad x_1}$
This corresponds to what is expected in terms of semantics. The `unify` function, given in [13], is readable, but the proofs of its soundness (the result of `unify` is a unifier) and completeness (whenever there is a unifier, `unify` finds it) took more than 4000 lines of code. Thanks to it, we are able to express the translation algorithm, also given in [13], which is sound and complete and handles all SPJ queries.

If the selection condition is an equality conjunction, a preprocessing step, `expand_query`, transforms it into a sequence of selections, whose conditions are equalities. Next, the translation yields either 1) an equivalent query encapsulated in the `TQ` constructor, 2) `EmptyRel`, when the original one has no solution, or 3) `NoTranslation`, when the input query is not SPJ.

The translation algorithm relies on several auxiliary functions. First, `fresh_row n r`, is used for the base case and generates a row, `Trow r fr`, tagged by relation name `r`. The function `fr` maps attributes to fresh variables, starting from index `n`. The second one, `rename t1 t2`, is used for selections with the condition `t1 = t2`. If `t1` and `t2` are distinct constants, the renaming returns `None`. If there exists a substitution `rho` that replaces `t1` by `t2` or vice-versa, avoiding to change constants into variables, it returns `Some rho`. The substition `rho` is then applied to the whole tableau. The only time when `unify` is needed is for joins. In this case, the translation is applied to both operands. Compatibility on common attributes is ensured by applying the resulting substitution to the whole query.

The following lemma states that the algorithm behaves as expected. Our formalization helped us in making precise the exact behavior of the translation algorithm. In the informal presentation taken from textbooks, an *underlying assumption* is made about *freshness of variables* for the base case, which is quite tedious to handle at the formal level. To the best of our knowledge, our algorithm is the *first* formally specified and fully proved for such a translation.

```
Lemma algebra_to_tableau_expand_is_complete :
  ∀ (q : query) (n : nat) (I : relname → setT),
  well_sorted_instance I →
  match algebra_to_tableau (S n) (expand_query q) with
    | TQ _ Ts  ⇒ ∀ t, is_a_solution I Ts t ⟷
                            t ∈ (eval_query I q)
    | EmptyRel ⇒ ∀ t, t ∈ (eval_query I q) → False
    | NoTranslation ⇒ translatable_q q = false
  end.
```

7.3. Data Integrity

Integrity constraints capture the semantics of data, as introduced in Section 4.3. To illustrate, we revisit our running example of a **Cinema** database (see Section 4.1). For the **Movies** relation, we may know that there is only one director associated with each movie title. Such a property is called a *functional dependency*, denoted $\{Title\} \hookrightarrow \{Director\}$, and refers to the fact that values of some tuple attributes uniquely determine other attribute values of that tuple. This type of dependency corresponds to the more general class of *equality generating dependencies*.

Let us further assume the relation: **Showings***(Theater, Screen, Title, Snack)*, containing a tuple *(th, sc, ti, sn)*, if the theater *th* is showing the movie *ti*, on the screen *sc* and offers snack *sn*. Intuitively, one would expect a certain independence between the *Screen-Title* attributes, on the one hand, and the *Snack* attribute, on the other, for a given *Theater* value. For instance, if *(Action Christine, 1, Casanova, Coffee)* and *(Action Christine, 2, M, Tea)* are in *Showings*, we also expect *(Action Christine, 1, Casanova, Tea)* and *(Action Christine, 2, M, Coffee)* to be present. Such a property is called a *multivalued dependency* and is denoted $\{Theater, Screen, Title\} \twoheadrightarrow \{Snack\}$. It refers to the fact that, if the values of some tuples t_1 and t_2 coincide for $\{Theater, Screen, Title\}$, then *Showings* also has to contain tuples obtained from t_1 and t_2, by swapping the corresponding *Snack* values. This type of dependency is *tuple generating*.

Equality and tuple generating dependencies fall under the yet wider class of *general dependencies*. This also captures inclusion dependencies, corresponding to the foreign key constraints, enforced by real database systems. As seen in Section 4.3, a central issue concerning dependencies is that of the so called *logical implication*, *i.e*, what other constraints can be inferred from a given set of constraints. We exemplified the deductive and procedural approaches to solving this problem, by presenting Armstrong's system - together with its extension with multivalued dependency inference rules - and the chase procedure. The former allows to deduce, in the functional (and multivalued) case, all solutions to the logical implication problem. As such, it is sound, complete and terminating. The latter allows to deduce logical implication solutions, in the general dependency setting. The chase is sound, but might not terminate in general.

In this section we present a Coq formalization of functional and multivalued dependencies, together with their inference procedures and corresponding characterizations. Also, we extend our modelization to the wider scope of general dependencies, formalize the chase and its corresponding characterization.

7.3.1. Logical Implication for Functional Dependencies

Syntactically, functional dependencies between attribute sets (`setA`) are modeled with the Coq inductive `fd`. The semantics `fd_sem` of a functional dependency `V ↪ W` is defined with respect to an instance, i.e, a set of tuples `I : setT`. Mirroring Definition 4.3.1,

this captures the fact that any two instance tuples `t1` and `t2` whose projections agree on `V`, also have to have their projections agree on `W`.

```
Inductive fd : Type := FD : setA → setA → fd.
Notation "V '↪' W" := (FD V W).

(* tuples t1 and t2 agree on a set of attributes S *)
Definition tupl_agree (t1 : tuple) (t2 : tuple) (S : setA) :=
  ∀ a, a ∈ S → dot T t1 a = dot T t2 a.

(* I ⊨ d, where d is a functional dependency *)
Definition fd_sem (I : setT) (d : fd) :=
  match d with
    | V ↪ W ⇒ ∀ t1 t2, t1 ∈ I → t2 ∈ I →
                tupl_agree t1 t2 V → tupl_agree t1 t2 W.
  end.
```

Armstrong's inference system $\mathcal{A}$ is modeled via the `dtree` inductive definition, representing a derivation tree, whose branches are the inference rules in Figure 4.2. To these, we added the `FD_ax` rule, for deriving dependencies already in the context. We denote the type of functional dependency sets by `setF`.

```
(* F ⊢ X ↪ Y *)
Inductive dtree (F : setF) : fd → Type :=
 | FD_ax    : ∀ X Y, (X ↪ Y) ∈ F → dtree F (X ↪ Y)
 | FD_refl  : ∀ X Y, Y ⊆ X → dtree F (X ↪ Y)
 | FD_aug   : ∀ X Y Z XZ YZ, XZ =set= (X ∪ Z) → YZ =set= (Y ∪ Z) →
              dtree F (X ↪ Y) → dtree F (XZ ↪ YZ)
 | FD_trans : ∀ X Y Y' Z, Y =set= Y' →
              dtree F (X ↪ Y) → dtree F (Y' ↪ Z) →
              dtree F (X ↪ Z).
```

The soundness of Armstrong's system, i.e, `Armstrong_soundness`, corresponds to Theorem 4.3.7. Its Coq proof is similar to the paper one and follows by induction on the derivation tree. We show that, for any set of functional dependencies `F` and for any functional dependency `d`, if `d` is syntactically derivable from `F`, i.e, `dtree F d`, then it is logically implied by `F` as well, i.e, `F ⊨ d`. The conclusion is equivalent to showing that, if an arbitrary instance `I` implies all dependencies `d'` in `F`, then `I` implies `d` as well.

```
(* F ⊢ d then F ⊨ d, i.e, ∀ I, I ⊨ F implies I ⊨ d *)
Theorem Armstrong_soundness : ∀ F d I,
  dtree F D → (∀ d', d' ∈ F → fd_sem I d') → fd_sem I d.
```

Next, the completeness proof, corresponding to Theorem 4.3.9, borrows from [80] the central idea of building a *model* `M`. Given a set of dependencies F and a set of attributes X, `M` consists of two tuples `t0` and `t1`, which only agree on the closure attribute set $[X]_F^+$. The constructive proof of completeness is based on the fact that, if $F \models X \hookrightarrow Y$, since M is a model of F, then `M` is a model of $X \hookrightarrow Y$.

Interestingly, while for soundness the hypotheses did not make any assumption on the finiteness of the attribute universe, for the completeness, this assumption was needed. All intermediate lemmas are given in [13] and the main theorem explicitly mentions the fact that all sets of attributes are included in the finite universe `U` and that the values zero and one are distinct.

```
Lemma Armstrong_completeness : ∀ U F X Y, X ⊆ U → Y ⊆ U →
  (∀ I, (∀ t, t ∈ I → support T t =[set] U) →
        (∀ f, f ∈ F → fd_sem I f) → fd_sem I (X ↪ Y)) →
  (dtree F (X ↪ Y)).
```

7.3.2. Logical Implication for Multivalued Dependencies

To also account for the tuple generating dependencies from Section 4.3.2, we extend the previous `fd` inductive with the `MD` constructor. The resulting `dep` inductive encodes the type of functional and multivalued dependencies. Similarly, we enrich the dependency inference inductive `dtree` with the corresponding rules for multivalued dependencies, described in Figure 5.2.2. The resulting `dtree_ext` inductive captures the fact that a given dependency `d : dep` can be inferred from a set of functional and multivalued dependencies `F : Fset.set FMD`.

```
Hypothesis U  : Fset.set FAttr.
Inductive dep : Type :=
 | FD : Fset.set FAttr → Fset.set FAttr → dep
 | MD : Fset.set FAttr → Fset.set FAttr → dep.
 Notation "V ↠ W" := (MD V W).

Definition OMD : Eoset.Rcd dep. (* ... *)
Definition FMD := FiniteSet.build_fset OMD.

(* F ⊢ X ↪ Y or F ⊢ X ↠ Y *)
Inductive dtree_ext : Fset.set FMD → dep → Prop :=
 | FD_ax    : ∀ F X Y, (X ↪ Y) ∈ F → dtree_ext F (X ↪ Y)
 | FD_refl  : ∀ F X Y, Y ⊆ X → dtree_ext F (X ↪ Y)
 | FD_aug   : ∀ F X Y Z, dtree_ext F (X ↪ Y) →
              dtree_ext F ((X ∪ Z) ↪ (Y ∪ Z))
 | FD_trans : ∀ F X Y Z, dtree_ext F (X ↪ Y) →
```

```
                dtree_ext F (Y ↪ Z) → dtree_ext F (X ↪ Z)
  | MD_compl : ∀ F X Y, X ⊆ U → Y ⊆ U → dtree_ext F (X ↠ Y) →
                dtree_ext F (X ↠ (U \ (X ∪ Y)))
  | MD_aug   : ∀ F X Y Z, dtree_ext F (X ↠ Y) → Z ⊆ U →
                dtree_ext F ((X ∪ Z) ↠ (Y ∪ Z))
  | MD_trans : ∀ F X Y Z, X ⊆ U → Z ⊆ U →
                dtree_ext F (X ↠ Y) → dtree_ext F (Y ↠ Z) →
                dtree_ext F (X ↠ (Z \ Y))
  | FM_conv  : ∀ F X Y, dtree_ext F (X ↪ Y) → dtree_ext F (X ↠ Y)
  | FM_inter : ∀ F X Y Z, dtree_ext F (X ↠ Y) → Z ⊆ U →
                dtree_ext F ((X ∪ Y) ↪ Z) →
                dtree_ext F (X ↪ (Z \ Y)).
```

Note that, as we only proved the soundness of `dtree_ext` - and not also its completeness, as in Section 7.3.1 - in defining constructors `FD_aug` and `FD_trans`, a less general form than that for their previous counterparts sufficed. Also, we did not need a constructor for the multivalued dependency reflexivity rule MVD_1 - present in the inference system from Figure 4.4 - as it follows from `FM_conv` and `FD_refl`.

Next, we give the semantics `mvd_sem` of a multivalued dependency $V \twoheadrightarrow W$ with respect to an instance `I : setT`. This mirrors Definition 4.3.10 and corresponds to the schematic representation in Figure 7.3. The tuple `t'` is obtained swapping `t1` and `t2` in `mvd_sem`.

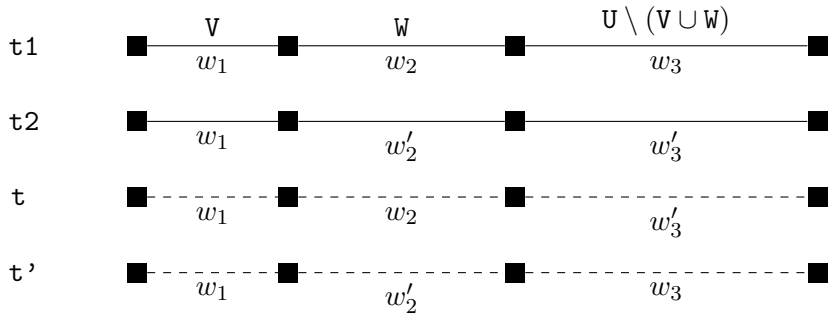


Figure 7.3.: Multivalued Dependency $V \twoheadrightarrow W$ over the attribute set U

```
Definition mvd_sem (I : setT) (V W : setA) :=
  ∀ t1 t2, t1 ∈ I → t2 ∈ r → tupl_agree t1 t2 V →
  ∃ t, t ∈ I ⋀ tupl_agree t t1 V ⋀ tupl_agree t t1 W
             ⋀ tupl_agree t t2 (U \ (V ∪ W)).
```

Next, we encode the notion of satisfiability of a functional or multivalued dependency `d`, using `fd_sem` and `mvd_sem`, depending on the nature of `d`.

```
(* I ⊨ d, where d is a functional or multivaled dependency *)
Definition sat_dep I d :=
  match d with
    | X ↪ Y ⇒  fd_sem I X Y
    | X ↠ Y ⇒ mvd_sem I X Y
  end.
```

Finally, we can state the soundness of the inference system `dtree_ext`, for functional and multivalued dependencies, corresponding to Theorem 4.3.11. The Coq proof is analogous to the one presented in Section 4.3.2.

```
(* F ⊢ X ↠ Y then F ⊨ X ↠ Y *)
Lemma Mvd_soundness : ∀ F dep I,
  dtree_ext F d → (∀ d', d' ∈ F → sat_dep I d') → sat_dep I d.
```

7.3.3. Logical Implication for General Dependencies

Constraints described in textbooks (functional, join or inclusion dependencies) fall into the wider category of *general dependencies*. Recall from Section 4.3.3 that these are first-order logic sentences of the form:

$$\forall x_1 \ldots \forall x_n(\phi(x_1, \ldots, x_n) \Rightarrow \exists z_1 \ldots \exists z_k \psi(x_1, \ldots, x_n, z_1, \ldots, z_k)),$$

where ϕ is a (possibly empty) conjunction of atoms and ψ an atom. In both ϕ and ψ, one finds relation atoms of the form $r(w_1, \ldots, w_l)$ and equality atoms of the form $w = w'$, where each of the $w, w', w_1, \ldots, w_l$ is a variable or a constant. Inclusion dependencies can be expressed by $\forall x_1 \ldots \forall x_n(r_1(x_1, \ldots, x_n) \Rightarrow r_2(x_1, \ldots, x_n))$. According to textbooks, the semantics of such formulas is the natural one. There is a strong relationship between general dependencies and tableaux, which provides a convenient notation for expressing and working with dependencies.

For example, the functional dependency $A \hookrightarrow B$ on the relation $r(A, B)$, is represented by the formula: $\forall v \forall v_1 \forall v_2,\ r(v, v_1) \wedge r(v, v_2) \Rightarrow v_1 = v_2$ and by the conjunctive query:

A	B	
v	v_1	r
v	v_2	r
$v_1 = v_2$		

When the right part of the implication is a relation predicate, the last line is a summary and such dependencies are referred to as *tuple generating*, while the other ones are referred as *equality generating*. We model this by the following inductive definition of

`gd`. According to whether ϕ is a relation predicate or an equality, we use the `TupleGen` and `EqGen` constructors.

```
Notation setR := (Fset.set (Ftrow T DBS)).

(* row equivalence *)
Notation "s1 '=^r=' s2" := (Fset.elt_compare Ftrow s1 s2 = Eq).

Inductive gd :=
 | TupleGen : setR → trow → gd
 | EqGen    : setR → tvar → tvar → gd.
```

The natural semantics is provided by:

```
Inductive gd_sem : gd → setT → Prop :=
 | TupleGenSem :
     ∀ (SR : setR) (s : trow) (I : setT),
        (∀ (ν : valuation),
           (∀ x, x ∈ SR → (ν ⟦x⟧_t) ∈ I) →
            ∃ ν_e, (∀ x, x ∈ variables_tableau SR → ν_e ⟦x⟧ = ν ⟦x⟧)
                  ⋀ ν_e ⟦s⟧_t ∈ I) → gd_sem (TupleGen SR s) I
 | EqGenSem :
     ∀ (SR : setR) x1 x2 (I : setT),
        (∀ (ν : valuation),
           (∀ x, x ∈ SR → ν ⟦x⟧_t ∈ I) →
            ν ⟦x1⟧ = ν ⟦x2⟧) → gd_sem (EqGen SR x1 x2) I.
```

The only subtle point in this definition is that it is stated for tableaux, but corresponds exactly to the semantics of logical formulas. Due to the particular form of the latter, given a valuation ν assigning values to the x's we extend it by ν_e over the existentially quantified z's.

The Chase We *tried to formalize* what is very informally provided by textbooks with the following inference rules.

Let d and d' be respectively $\forall \vec{x},\ \phi(\vec{x}) \Rightarrow \exists \vec{z},\ \psi(\vec{x} \cup \vec{z})$ and $\forall \vec{x'},\ \phi'(\vec{x'}) \Rightarrow \exists \vec{z'},\ \psi'(\vec{x'} \cup \vec{z'})$. For applying d' to d, we first need to find a mapping ν, such that $\nu(\phi'(\vec{x'}))$ - seen as a set of atoms - is a subset of $\phi(\vec{x})$. Depending on the form of ψ', we get:

1. if $\psi' \equiv y'_1 = y'_2$ and ρ is the renaming: $\{\nu(y'_2) \mapsto \nu(y'_1)\}$
   then $chase(d, d')$ is $\forall \vec{x},\ \rho(\phi(\vec{x}) \Rightarrow \exists \vec{z},\ \psi(\vec{x} \cup \vec{z}))$.
2. if $\psi' \equiv r'(\vec{y'})$ then $chase(d, d')$ is $\forall \vec{x},\ \phi(\vec{x}) \wedge \nu(r'(\vec{y'})) \Rightarrow \exists \vec{z},\ \psi(\vec{x} \cup \vec{z})$.

However, the above version is faulty, due to variable capture for $\nu(r'(\vec{y'}))$ by $\forall\vec{x}$, which naturally arose in the second case, as shown by the following counterexample.

Let d be $\forall y\ z,\ r(y,y,z) \Rightarrow r(y,y,y)$ and d' be $\forall x\ y,\ r(x,x,y) \Rightarrow \exists z,\ r(x,z,x)$.
We recall the main idea behind logical implication in the context of the chase procedure, namely that, $\forall I$, if $I \models d'$ and $I \models chase(d,d')$, then, trivially, $d' \models d$. With the mapping $\nu = \{x \mapsto y, y \mapsto z\}$, the above definition yields:

$$chase(d,d') \equiv \forall y\ z,\ r(y,y,z) \wedge r(y,z,y) \Rightarrow r(y,y,y).$$

Consider the instance $\mathcal{I} = \{(a,a,b),(a,c,a)\}$. We have $\mathcal{I} \models d'$, and $\mathcal{I} \models chase(d,d')$, since there is no μ, such that $\mu(y,y,z) \in \mathcal{I} \wedge \mu(y,z,y) \in \mathcal{I}$. But $\mathcal{I} \not\models d$, as shown by $\mu_1 = \{y \mapsto a, z \mapsto b\}$, since $\mu_1(y,y,z) = (a,a,b) \in \mathcal{I}$ and $\mu_1(y,y,y) = (a,a,a) \notin \mathcal{I}$.

This counterexample does not affect the essence of the theorem, but emphasizes the fact that humans naturally perform α-conversion in order to avoid capture; therefore, when defining the chase in Coq, this had to be explicitly taken into account.

Since variables (in the gd's) are indexed by integers, in order to avoid captures, we generate fresh variables for renaming, starting from the maximum index of all variables in the constraints, computed thanks to the `max_var_chase` function. Then, `avoid_capture_trow max_n phi' psi'` computes a renaming for the variables which are in `psi'` and not in `phi'`. The chase may yield three different results: the first one is when there is at least one ν producing a new constraint, the second captures the fact that no such mappings exist and the third corresponds to the fact that the current dependency tries to identify two distinct constants. There is one further subtle point to detail.

Given a pair of dependencies, there may exist several mappings ν, thus, in order to avoid the design of a lazy matching function, we chose to apply them simultaneously. The first case applies an equality generating dependency `EqGen SR x1 x2`. It consists of iterating the replacement of `nu x1` by `nu x2`, for all such ν's. The second case applies a tuple generating dependency `TupleGen SR s`. In that case, we simply add all `nu s` to current tableau. The only point is to avoid capture for existential variables and also to avoid interference between the different mappings. This has the unfortunate consequence that the chase step given in [13], as well as the soundness proofs, are intricate.

As the chase terminates only for a specific class of dependencies (the one with no existential quantifiers), we defined a special "for loop" that iterates the application of a dependency set over d a fixed number of times. At this point, the algorithm stops with a (potentially) new dependency. If this dependency is trivial (*i.e.,* either of the form $\forall\vec{x},\ \phi(\vec{x}) \Rightarrow y = y$ or $\forall\vec{x},\ \phi(\vec{x}) \Rightarrow \exists\vec{z},\ \psi(\vec{x} \cup \vec{z})$, where there exists a substitution σ for z's, such that $\psi(\vec{x} \cup \vec{\sigma(z)})$ is an atom of $\phi(\vec{x})$), then the initial dependency set implies d.

Lastly, the soundness of the chase procedure is established by:

```
Inductive res : Type := Res : gd → res | NoProgress | Fail.

Definition var_in_query x SR :=
  match x with
  | Tvar _ ⇒ x ∈ variables_gd SR
  | Tval _ ⇒ True
  end.

Lemma chase_is_sound : ∀ ST n D d d',
  chase n d D = Res d' →
  match d with
    | TupleGen _ _ ⇒ True
    | EqGen SR1 x1 x2 ⇒ var_in_query x1 SR1 ⋀ var_in_query x2 SR1
  end → (∀ gd, List.In gd D → gd_sem gd ST) → gd_sem d' ST →
  gd_sem d ST.
```

Doing the proof, the main subtle point was to avoid variable capture through iteration. Again, it was during this proof step that we discovered that the textbooks were imprecise. The needed functions and technical lemmas are given in [13].

7.4. Discussion

Our formalization effort comprises two different aspects: one concerning modeling and the other, proving properties of different operators and algorithms. With respect to the proof aspects, we had to explicit technical points, such as freshness, unification and variable capture avoidance. While these are not new for Coq users or even for the functional programming community, they are worth precising for database theoreticians and practitioners. The main reason is that such aspects are neither mentioned in textbooks, nor do they appear explicitly in implementations (usually written in C). The real challenge was modeling. Our contribution, unlike, [41] is almost complete. We were able to model all these various aspects because our very first choices for attributes, tuples were adequate. Such choices were not trivial nor immediate and neither [60] nor [41] made them; hence, they never reached the generality we achieved. Obviously, once the right choices are done, the whole seems simple.

7.4.1. Contributions

In this chapter, we presented a Coq formalization of the relational model. The formalization consists of the following components:

- the relational data model (relations, tuples, etc.)
- a mechanization of integrity constraints (general dependencies) and their characteristic properties:

  - functional dependencies (subclass of the equality generating dependencies) and their inference system (Armstrong);
    proofs establishing the soundness and completeness of their logical implication
  - multivalued dependencies (subclass of the tuple generating dependencies) and their inference system (extended Armstrong);
    the soundness proof for logical implication of functional and multivalued dependencies considered together
  - general dependencies (equality and tuple generating) and their inference system (the chase);
    the soundness proof for logical implication of general dependencies

- mechanization of the main relational query languages:
  relational algebra and conjunctive queries
- proof of the main "database theorems":
  algebraic equivalences, the homomorphism theorem and conjunctive query minimization.

7.4.2. Lessons

In a first version of our development, we heavily used dependent types and proofs in types. In particular, they expressed that tuples and queries were well-typed by construction. But, we experienced a lot of problems with type conversion in proofs. In all algorithms given in this thesis, it is *crucial* to check equality (or congruence). In Coq one can only check equality between two terms which belong to the *same* Type. With dependent types, there are two possibilities: either to use type conversion or John Major equality (fortunately we fall in the decidable case). Both are very cumbersome. Moreover, in order to debug we needed to run the algorithms with well-typed terms (*i.e.*, with hand-written proofs embedded in types). The benefits of our approach are three (i) with it, it is easier and lighter to write algorithms and perform case analysis in proofs (ii) it is closer to main stream programming languages in which real systems are encoded (iii) it precisely allows to locate where well-typedness is needed. Surprisingly, we discovered that types, in the usual sense, were not useful, rather, the notion of *well-sortedness* was indeed crucial. This is an a posteriori justification of the fact that in all theoretical books values range in a unique domain. Specifying the main algorithms and proving the "database theorems" for tableaux and the chase led us to thoroughly make explicit some notions or definitions which were either unclear or at least very sloppy. For example, freshness or variables' capture are almost completely left aside in textbooks. However, such notions are *central to the correctness* of the results, as shown by our counter-example.

8. Formalization of Standard Datalog

In this chapter we discuss a formalization of standard (positive) Datalog, based on notions introduced in Chapter 5. The library consists of a mechanization of the language, of its semantics (model-theoretic and fixpoint), and of an inference engine, implementing the bottom-up evaluation heuristic. It contains proofs capturing the characteristic properties of all defined components. We briefly summarize below the main idea behind the functioning of the engine, which lies at the core of the development.

The inference engine essentially iterates a fixpoint operator, which is based on the implementation of immediate consequence (see Definition 5.2.9) through an abstract matching algorithm. The goal of the inference procedure is to build a model of an input standard Datalog program. To this end, the engine maintains a current interpretation - the "candidate model" - which it iteratively tries to "repair". The repair process first identifies clauses that violate satisfiability - i.e, whose ground instances have bodies that are satisfied by the current interpretation, but whose heads are not. The current interpretation is then "fixed", by adding to it the missing facts, corresponding to head groundings. This is realized using forward chain's *matching* algorithm, which computes all substitutions matching body atoms to interpretation facts. Since the safety condition ensures all head variables appear among those in the body, it follows that the resulting substitution is, indeed, grounding. Hence, applying it to the head produces a *new fact*. Once the current interpretation is "updated" with all facts that can be inferred in one forward chain step, the procedure is repeated, until a fixpoint is reached. We prove this to be a *minimal model* of the input program.

The chapter is organized as follows. We start with Section 8.1, by outlining the relevant modelization choices we made. Next, in Section 8.2 and Section 8.3, we detail the way in which we represented the language and its semantics. In Section 8.4, we present the base building blocks needed for defining bottom-up evaluation and, in Section 8.5, we identify relevant properties of the heuristic. We conclude in Section 8.6, with a characterization of the inference engine, as captured by the main proof, establishing its evaluation as *sound*, *complete* and *terminating* with a *minimal model* output.

8.1. Modelization Choices

In the formalizations presented, on the one hand in Chapter 7 and, on the other hand in Chapter 8 and Chapter 9, we explore two different modelization approaches. The first is based on having infinite (unrestricted) models, via simple datatype encodings and trailing well-formedness side-conditions, and, the second, on having finite (restricted)

models, via richer datatypes that obey well-formedness by construction. We argue that the logic programming setting, as represented by DATALOG, lends itself particularly well to the latter approach. This is due to the fact that the declarative nature of the language and its simple evaluation procedure favor a mathematical description. Such an approach is heavily based on the compositional properties of more complex algebraic structures and operators.

Next, we give an account of the main choices impacting the development of the standard DATALOG library. These correspond to the formalization of domains (Section 8.1.1), the choice of primitives (Section 8.1.2), the formalization of ground atoms (Section 8.1.3) and that of groundings and substitutions (Section 8.1.4).

8.1.1. Finite vs Infinite Domains

The fundamental choice we made in the formalization of standard DATALOG is to model the underlying database domains as finite sets. As such, we rely on the `finset` formalization of the theory of sets over finite types, provided by the MATHEMATICAL COMPONENTS library (see Chapter 2).

The fact that we can base our development on this premise, *without loss of generality*, is motivated by the following observation, whose proof we detail below.

Theorem 8.1.1. *Assume a safe, positive* DATALOG *program P. Also, let I be an arbitrary Herbrand interpretation of P, i.e, $I \subseteq \mathbf{B}(\mathbf{P})$[1]. The satisfiability of P with respect to I under (assignments mapping to) $adom(P)$, i.e, the active domain of P, coincides with validity under (assignments mapping to) larger domains $\mathcal{D}$, i.e, such that $\mathcal{D} \supseteq adom(P)$. This statement is expressible in logic form as:*

$$\forall P, \forall I, \forall \mathcal{D}, (\forall C, C \in P \Rightarrow V(head(C)) \subseteq V(body(C))) \Rightarrow I \subseteq \mathbf{B}(\mathbf{P}) \Rightarrow$$
$$\mathcal{D} \supseteq adom(P) \Rightarrow ((\forall \iota, \iota : \mathcal{X} \to adom(P) \Rightarrow I, \iota \models P) \Leftrightarrow (\forall \iota', \iota' : \mathcal{X} \to \mathcal{D} \Rightarrow I, \iota' \models P)).$$

Proof. Let P be a safe standard DATALOG program and $C = H \leftarrow \vec{B}$ be a safe clause, such that $C \in P$. Proving the above statement amounts to showing that:

$$(\forall \iota, \iota : \mathcal{X} \to adom(P),\ I, \iota \models C) \Leftrightarrow (\forall \iota', \iota' : \mathcal{X} \to \mathcal{D},\ I, \iota' \models C)$$

According to the definition of clause interpretation in Section 5.2, this reduces to:

$$(\forall \iota, \iota : \mathcal{X} \to adom(P),\ [\![\vec{B}]\!]^{I,\iota} = \bot \vee [\![H]\!]^{I,\iota} = \top) \Leftrightarrow$$
$$(\forall \iota', \iota' : \mathcal{X} \to \mathcal{D},\ [\![\vec{B}]\!]^{I,\iota'} = \bot \vee [\![H]\!]^{I,\iota'} = \top)$$

and, given the Herbrand semantics setting, to:

$$(\forall \iota, \iota : \mathcal{X} \to adom(P),\ \iota\vec{B} \not\subseteq I \vee \iota H \in I) \Leftrightarrow (\forall \iota', \iota' : \mathcal{X} \to \mathcal{D},\ \iota'\vec{B} \not\subseteq I \vee \iota' H \in I)$$

[1] Note that imposing this condition on models is not a strong restriction per-se, since larger models would only additionally contain "trash", i.e, facts that are not relevant to the program.

which is equivalent to:

$$(\forall \iota, \iota : \mathcal{X} \to adom(P),\ (\iota\vec{B} \subseteq I \Rightarrow \iota H \in I)) \Leftrightarrow$$
$$(\forall \iota', \iota' : \mathcal{X} \to \mathcal{D},\ (\iota'\vec{B} \subseteq I \Rightarrow \iota' H \in I))$$

$\Leftarrow$ Trivial, since, from $\mathcal{D} \supseteq adom(P)$, it follows that we can view an assignment $\iota : \mathcal{X} \to adom(P)$ as a more general one $\iota' : \mathcal{X} \to \mathcal{D}$.

$\Rightarrow$ Let $\iota' : \mathcal{X} \to \mathcal{D}$ and $\iota'\vec{B} \subseteq I$. Since $I \subseteq B(P)$, where $B(P)$ only contains constants in $adom(P)$, it follows from $\iota'\vec{B} \subseteq I$ that $\iota' V(\vec{B}) \subseteq adom(P)$ (1). As the clause safety condition ensures $V(H) \subseteq V(\vec{B})$, we have $\iota' V(H) \subseteq \iota' V(\vec{B})$ (2). From (1) and (2), by transitivity, $\iota' V(H) \subseteq adom(P)$ (3).
Let $\iota'' : \mathcal{X} \to adom(P)$ be an assignment agreeing with ι' on $V(\vec{B})$ (4). From (1), we have $\iota'' V(\vec{B}) \subseteq adom(P)$. Hence, we can instantiate with ι'' the hypothesis: $\forall \iota, \iota : \mathcal{X} \to adom(P) \Rightarrow (\iota\vec{B} \subseteq I \Rightarrow \iota H \in I)$. We obtain $\iota''\vec{B} \subseteq I \Rightarrow \iota''(H) \in I$ (5).
From (4) and $\iota'\vec{B} \subseteq I$, we have $\iota''\vec{B} \subseteq I$, which in turn, from (5), entails $\iota''(H) \in I$.
Since $V(H) \subseteq V(\vec{B})$, from (4), it follows that ι'' and ι' agree on $V(H)$.
Consequently, $\iota' H \in I$.

□

8.1.2. Separating Syntax and Semantics Objects

We found it easier to advance in constructing the development, by *explicitly* separating the objects that belong to the syntax level of the language from those that *implicitly* belong to the semantics level, i.e that do not contain variables. Hence, we provide different types for (open) atoms and ground atoms (`atom` and `gatom`) and for substitutions and groundings, i.e, closed substitutions, (`sub` and `gr`). The main motivation is that we did not want to reason about variables, every time we assumed an interpretation, as this is more cumbersome and increases proof size. However, we envisage setting up a more uniform treatment of these primitives, as an interface to the existing development.

8.1.3. Modeling Ground Atoms

The main diverging point between the *infinite*-state and the *finite*-state modelisation approaches is given by the way in which (ground) atoms are represented.

In the first scenario, ground atoms are defined through an inductive type `gatom`, packing a predicate symbol type and the type of its arguments, i.e, a list of constants.

```
Inductive gatom := GAtom of symtype & seq constant.
```

In the second scenario, they are defined via a record packing a base ground atom type `raw_gatom`, encoded exactly as the `gatom` before, and a boolean well-formedness condition `wf_raw_gatom`, imposing the number of arguments correspond to the symbol arity.

```
Inductive raw_gatom := RawGAtom of symtype & seq constant.
Definition wf_gatom rga :=
  size (arg_raw_gatom rga) == arity (sym_raw_gatom rga).

Structure gatom := GAtom {rga :> raw_gatom; _ : wf_raw_gatom rga}.
```

It is important to note that, while working with dependent types may render proofs more cumbersome in general, these are much simpler to maneuver when built with boolean subtyping predicates. Indeed, such predicates are innately *proof-irrelevant* [46], since there is *only* one boolean equality proof, thus any two are trivially equal. The corresponding subtype projection `rga` is thus injective, i.e, `rga ga1 = rga ga2` implies `ga1 = ga2`. That is to say, checking equality of well-formed raw ground atoms can be reduced to checking equality of the underlying raw ground atoms.

In the rest of the chapter, we will rely on the second modelization of ground atoms. As detailed in Section 8.3.4, a key step in our development is establishing the finiteness of their type. To prove soundness and completeness of the inference, we take advantage of the library support `finType` provides in the finite setting.

We comment on the main differences with respect to the *infinite*-state modelisation in Section 8.7. There, we discuss an alternative development we built, which does not make the finiteness assumption. This is based on Cyril Cohen's `finmap` library[2].

8.1.4. Modeling Groundings and Substitutions

Recall that a DATALOG program P does not contain function symbols and that, due to the safety condition, all its facts are ground. To build a model for P, we rely on the immediate consequence operator, which computes new facts from relevant groundings for clause heads. As such, it implements an algorithm *matching* program atoms to facts [3]. This, in turn, is based on *matching* variables to constants. Consequently, while in general (see Section 3.1), substitutions have type $\mathcal{X} \to T_\Sigma(\mathcal{X})$, for the purposes of our formalization it is pertinent and sufficent to restrict (ground) substitutions to being mappings from variables to constants, i.e, of type $\mathcal{X} \to \mathcal{C}$.

The only difference between groundings and substitutions is that the former is a total function, while the latter is partial. The design choices made in representing these mappings were influenced by the **finiteness** of their domain type and, in the case of substitutions, by the **partiality** of the codomain.

Indeed, as we conveniently expressed the type of variables via a finite type (more specifically, an ordinal type - cf. Section 2.2 - bounded by the number of program variables)

[2] `https://github.com/Barbichu/finmap`
[3] Note hence that we can restrict ourself to a more particular case than that of unification.

both mappings can be encoded through **finitely-supported functions** (`ffun`), as defined in the `finfun` library.

Essentially, the idea behind this special SSReflect representation is that an arbitrary function $f : A \to B$ over a finite domain A, can be seen as a *tabulation*. Indeed the corresponding `fun_to_ffun` definition is stated constructing a canonical function tuple `codomain_tuple`, whose length is given by the number of inhabitants of A and whose elements are inhabitants of B. The corresponding SSReflect expression is `#|A|.-tuple B` and abbreviates to `{ffun A → B}`. Note that the `tuple` type is not the same as that used in the Coq formalization of the relational model, from Chapter 7.

```
Canonical codom_tuple : #|A|.-tuple B := [tuple of codom f].
Definition fun_to_ffun (A : finType) (B : Type) (f : A → B)
  : {ffun A → B} := Finfun (codom_tuple f).
```

The main advantage of seeing groundings and substitutions as instances of this type, is that they now benefit **automatically** from the `ffunP` **extensional equality** property.

```
Definition eqfun (f g : B → A) : Prop := ∀x, f x = g x.

Notation "f1 =1 f2" := (eqfun f1 f2)
  (at level 70, no associativity) : fun_scope.

(* for finitely-supported functions f1 and f2,
   extensional equality is equivalent to functional equality *)
Lemma ffunP (A : finType) (B : Type) (f1 f2 : {ffun A → B}) :
  f1 =1 f2 ↔ f1 = f2.
```

Moreover, we can easily coerce finitely-supported functions into regular ones, as illustrated below by `ffun_to_fun`, thus still being able to use all the common functional properties, in particular injectivity. This is realized by transforming a finitely-supported function `f` into a regular function `ffun_to_fun f`, which associates to each element `x` of the domain, the value `enum_rank x` at the corresponding index of the function's tabulation tuple `fgraph f`.

```
Definition ffun_to_fun (A : finType) (B : Type) (f : {ffun A → B})
  := fun (x : A) ⇒ tnth (fgraph f) (enum_rank x).
```

Due to the internals of the SSReflect tuple representation (see Section 2), we can also conveniently use finitely-supported functions as lists of pairings. Thus, this modeling approach subsumes the one taken in Section 7. Lastly, note that, when assuming a finite type for constants - by only considering those in the active domain (as we will in Section 9) - the type of substitutions becomes finitely-enumerable. Hence, given a

decidable predicate P on substitutions, $\forall x, Px$ also becomes decidable.

8.2. Language Representation

We overview the formalization of the *syntax* of a positive DATALOG program: namely, of its base *signature* (Section 8.2.1), of its (ground and open) *primitives* (Section 8.2.2 and Section 8.2.3) and of its *safety* restriction (Section 9.1.1).

8.2.1. Program Signature

As detailed in Section 8.1, we assume *finite types* for predicate symbols (`symtype`) and constants (`constype`), as well as a *finitely-supported function* for symbol arity.

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

8.2.2. Ground Primitives

The choices to define ground primitive types *separately* from the non-ground ones, as well as that concerning the type of ground atoms are explained in Section 8.1.

Ground Atoms Ground atoms are records packing a ground atom `raw_gatom` and a well-formedness condition `wf_raw_gatom`, ensuring the symbol arity matches the number of arguments. Recall that the latter is needed in order to ensure finiteness of the `gatom` type, as we only allow a finite number of arguments. Accessing the well-formedness property of a given ground atom `ga` is done by casing. Indeed, in accordance with the MATHEMATICAL COMPONENTS methodology, there is no named projector. This choice is motivated by robustness purposes, as not to depend on the internal representation.

```
Inductive raw_gatom := RawGAtom of symtype & seq constant.
Definition wf_raw_gatom ga :=
  size (arg_raw_gatom rga) == arity (sym_raw_gatom rga).

Structure gatom := GAtom {rga :> raw_gatom; _ : wf_raw_gatom rga}.
```

In the corresponding COQ proofs, we denote ground atoms with the bar notation, e.g, a ground body atom is represented as $\bar{B}$. As we will see in Section 8.2.3, a ground atom $\bar{B}$ can be lifted to an atom type, denoted as $\ulcorner \bar{B} \urcorner$, using the `to_atom` function.

In order to implement class hierarchy instances, e.g, for `eqType` and `choiceType`, we will relate our `raw_gatom` type to one that possesses them, namely its underlying pair

type, containing a symbol and a list of constants. Consequently, we define the mappings `raw_gatom_rep` and `raw_gatom_pre` that destruct and, respectively, construct a `raw_gatom` from such a pair. As symbols and constants already have the right instances, Coq will automatically infer corresponding ones for pairs and lists.

```
Definition raw_gatom_rep l := let: RawGAtom s a := l in (s, a).
Definition raw_gatom_pre l := let: (s, a) := l in RawGAtom s a.
```

We then prove the `raw_gatom_repK` cancelation lemma, stating that `raw_gatom_pre` is the left inverse of `raw_gatom_rep`. The lemma is stated using `ssrfun`'s `cancel` construct, whose definition we also give below. Since left-invertible functions are injective, we thus establish an embedding of our `raw_gatom` type into a simpler structure, inheriting all the properties of symbols and lists of constants.

```
Variables (A B : Type) (f : A → B) (g : B → A).

(* cancel f g ↔ g is a left inverse of f *)
Definition cancel f g := ∀x, g (f x) = x.

Lemma raw_gatom_repK : cancel raw_gatom_rep raw_gatom_pre.
Proof. by case. Qed.
```

In particular, due to ssreflect's canonical structure mechanism, briefly overviewed in Section 8.1, the decidable equality and choice type properties can be transferred to (raw) ground atoms. This is achieved by the `CanxxMixin` family of lemmas that build proper instances from injection proofs.

```
Canonical raw_gatom_eqType :=
  Eval hnf in EqType raw_gatom (CanEqMixin raw_gatom_repK).
Canonical raw_gatom_choiceType :=
  Eval hnf in ChoiceType raw_gatom (CanChoiceMixin raw_gatom_repK).
```

As explained in [35], ssreflect does not automate class inheritence for canonicals, so we declare separate ones corresponding to `eqType` and `choiceType`. Since `gatom` is a subtype of `raw_gatom`, it will inherit all the classes of this base type.

Defining an instance of ground atoms for the finite type class `finType` is more complex than shown previously for `eqType` and `choiceType`. The reason for this is that we have to bound the largest possible ground atom occuring in our program; this in turn means determining the maximal arity `max_ar` of the program symbols. Once we know `max_ar`, together with the assumption of finitely many constants, we obtain the required bound. All the ground atoms can thus be embedded in the finite type `(symtype * max_ar.-`

`tuple constant)`. Finally, the last step is providing a refinement of this type with the actual arity `x` of type `'I_(max_ar.+1)`.

In this case, we exhibit an embedding from `gatom` to a finite type `gatom_enc`. This encodes ground atoms as a pairing of a symbol type and a constant tuple of bounded length `'I_(max_ar.+1)`, whose type is inferred as finite.

```
Notation max_ar := (\max_(s in symtype) arity s).
Notation gatom_enc :=
  ({x : 'I_(max_ar.+1) & (symtype * x.-tuple constant)%type}).
```

A ground atom `ga` can be injected into `gatom_enc`, using the existential constructor for dependent pair types, i.e, `existT`. Indeed, since `ga` packs an underlying raw ground atom `rga` and its well-formedness proof `wf_rga`, we can build a `gatom_enc`. This is done by directly providing `sym_raw_gatom rga`, the symbol of the raw atom `rga` and by constructing a tuple from `wf_rga`. The proof `max_ar_bound`, for the boundedness of the corresponding tuple length ordinal, trivially relies on the `leq_bigmax_cond` lemma. This states the soundness of maximum value computation, using the `\max` big operator, for values satisying a given predicate.

```
Definition gatom_fenc (ga : gatom) : gatom_enc :=
  let: GAtom rga wf_rga := ga in
  existT _ (Ordinal (max_ar_bound (sym_raw_gatom rga)))
           (sym_raw_gatom rga, Tuple wf_rga).
```

Conversely, we can convert a ground atom encoding into a ground atom.

```
Definition fenc_gatom (e : gatom_enc): option gatom.
case: e ⇒ x [s];
case: (val x == arity s) / eqP ⇒ [→ | _] [tup proof];
[exact: (Some (@GAtom (RawGAtom s tup) proof)) | exact/None].
Defined.
```

To prove that the function `fenc_gatom` is a partial left-inverse of `gatom_fenc`, we apply casing, equality reflection and proof-irelevence, i.e, `eq_axiomK`.

```
Lemma fenc_gatomK : pcancel gatom_fenc fenc_gatom.
Proof.
by case⇒ [[? ?] ?] /=; case: eqP ⇒ // ?; rewrite !eq_axiomK. Qed.
```

Note that, to embed `gatom` into `gatom_enc`, we use the partial cancelation operator `pcancel`, defined below.

```
Variable (A B : Type) (f : A → B) (g : B → option A)

(* pcancel f g ↔ g is a partial left inverse of f *)
Definition pcancel f g := ∀x, g (f x) = Some x
```

Finallly, ee derive a finite type for gatoms through the `PcanFinMixin`, using the `fenc_gatomK` cancelation proof above.

```
Canonical gatom_finType :=
  Eval hnf in FinType gatom (PcanFinMixin fenc_gatomK).
```

Ground Clauses Ground clauses are defined through the `gclause` inductive, whose constructor packs two components: a distinguished ground atom, i.e the clause head, and a list of ground atoms, i.e the clause body.

```
Inductive gclause := GClause of gatom & seq gatom.
```

The definition of the class hierarchy instances for `gclause` follows the exact same pattern as above and is thus omitted for brevity.

8.2.3. Open Primitives

Terms Terms are formalized with an inductive joining 1) variables, which have an ordinal type `'I_n`, bounded by a maximal value n, computable from a given program, and 2) constants.

```
(* maximal number of program variables*)
Variable n : nat.

Inductive term : Type :=
| Var of 'I_n
| Val of constant.
```

Note that we have chosen to bound the number of program variables, in order to take advantage of a more convenient substitution theory. In no way is the bound essential to the development.

As before, we inject a term into its corresponding Σ-type representation `('I_n + constant)`, via the left cancelation lemma `term_repK`. Using this, we can then provide an instance of canonical terms with decidable equality.

Atoms Atoms are defined similarly to their ground counterpart in Section 8.2.2, as records packing an atom base type `raw_atom` and a well-formedness condition `wf_atom`. The point of maintaining the well-formedness condition over atoms is to ensure that grounding them will always produce well-formed ground atoms.

```
Inductive raw_atom := RawAtom of symtype & seq term.

Definition sym_atom a := let: RawAtom s_a _     := a in s_a.
Definition arg_atom a := let: RawAtom _ arg_a := a in arg_a.

Definition wf_atom a := size (arg_atom a) == arity (sym_atom a).

Structure atom := Atom {ua :> raw_atom; _ : wf_atom ua}.
```

The set of atom variables is the union of argument variables of the underlying `raw_atom`.

```
Definition term_vars t := 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 := raw_atom_vars a.
```

Note that Coq will insert the proper coercion from `atom` to `raw_atom` in `atom_vars`.

Programs Clauses are represented by an inductive grouping the head atom and the list of atoms in the body. Programs are lists of clauses.

```
Inductive clause   := Clause of atom & seq atom.
Definition program := seq clause.
```

8.2.4. Safety Condition

As explained in Section 5.1, a *safety condition*, encoded by `safe_prog`, is imposed on Datalog programs, to ensure computed models are finite. Specifically, this refers to the fact that, for each clause, the head variables are required to appear among those in the body.

```
Definition tail_vars tl := \bigcup_(t ← tl) atom_vars t.
Definition safe_cl cl   :=
  atom_vars (head_cl cl) ⊆ tail_vars (body_cl cl).
Definition safe_prog p  := all safe_cl p.
```

8.3. Semantics Representation

We first discuss groundings and substitutions in Section 8.3.1 and Section 8.3.2 and, in Section 8.3.3, how we relate the two. We define the semantics of a standard DATALOG program in Section 8.3.4.

Note that groundings and substitutions are modeled via *finitely-supported functions* from variables to constants, as detailed in Section 8.1.

```
Definition gr  := {ffun 'I_n → constant}.
Definition sub := {ffun 'I_n → option constant}.
```

Every grounding can be coerced to a substitution and, conversely, substitutions can be turned into groundings, by padding with a default element. It is often the case, in the proofs later presented in this chapter, that we want to prove that applying a substitution to an atom will produce a ground atom; it is fairly convenient to obtain such a ground atom from the grounding associated to the original substitution. In the mathematical presentation of the relevant COQ proofs, we will signal these transformations as follows: if ν `:` `gr`, then $\hat{\nu}$ `:` `sub` and, if σ `:` `sub`, then σ_{def} `:` `gr`.

8.3.1. Groundings

We define the application of a grounding ν to terms, atoms and clauses.

Term Grounding The grounding `gr_term` of a term `t` with a grounding ν is given by case analysis: if `t` is defined by a variable `v`, we return the constant resulting from applying ν to `v` and, if `t` is defined by a constant, we leave `t` unchanged.

```
Definition gr_term ν t :=
  match t with
    | Var v ⇒ ν v
    | Val c ⇒ c
  end.
```

Atom Grounding To define the grounding `gr_atom` of an atom `a` with a grounding ν, it suffices to provide a well-formedness proof for the resulting ground atom, as the latter can be automatically inferred. As such, we declare the *grounding* `gr_raw_atom` of the underlying `raw_atom` and prove its *well-formedness*, i.e that the size of its argument list matches its symbol arity.

Grounding a raw-atom `ra` follows from wrapping in a `RawGAtom` constructor: 1) the raw-atom's symbol `sym_atom ra` and 2) its argument grounding obtained by mapping the `gr_term` ν function on each term.

```
Definition gr_raw_atom ν ra :=
  RawGAtom (sym_atom ra) [seq gr_term ν x | x ← arg_atom ra]
```

The corresponding well-formedness is directly derivable from the `size_map` lemma stating list size is preserved by function mapping.

```
Definition gr_atom_proof ν a : wf_gatom (gr_raw_atom ν a).
Definition gr_atom ν a := GAtom (gr_atom_proof ν a).
```

Clause Grounding Clause grounding is given by applying the `GClause` constructor directly to the head grounding and to the body grounding, obtained mapping the atom grounding function.

```
Definition gr_cl ν cl :=
  GClause (gr_atom ν (head_cl cl))
          [seq gr_atom ν x | x ← body_cl cl].
```

8.3.2. Substitutions

We define the application of a substitution σ to terms, atoms and clauses, as well as an ordering on substitutions.

Term Substitution Applying a substitution σ to a term t is done by a term matching returning either the corresponding value in σ or leaving the term unchanged, if there is no corresponding value or if the term is constant.

```
Definition sterm σ t :=
  match t with
   | Val d ⇒ Val d
   | Var v ⇒ if σ v is Some d
             then Val d
             else Var v
  end.
```

Atom Substitution The definition of atom substitution is analogous to the above one for atom grounding, in that we only need to provide a well-formedness proof for `sraw_atom`, i.e the substituted underlying `raw_atom`. As before, constructing `sraw_atom` consists of wrapping the atom's symbol and the atom's substituted arguments, obtained by mapping the term substitution function `sterm` σ over the term list.

```
Definition sraw_atom σ ra :=
  RawAtom (sym_atom ra) [seq sterm σ x | x ← arg_atom ra].
```

The respective well-formedness proof is identical to `gr_atom_proof`, i.e, the proof of atom grounding well-formedness. Atom substitution is analogous to atom grounding.

```
Definition satom σ a := Atom (satom_proof σ a).
```

Substitution Membership A substitution binding `b` is a pair consisting of a variable and a constant, corresponding to the first and section projections `b.1` and `b.2`. The fact that a given binding belongs to a substitution is modeled with the `mem_binding` boolean predicate.

```
(* Binding b belongs to s *)
Definition mem_binding s b : bool := s b.1 == Some b.2.
```

We formalize membership in a substitution as a generic predicate (see Section 2), thanks to the `eqbind_class` definition. Consequently, we can implement the collective predicate interface `sub_of_eqbind` for substitutions. As such, we extend the generic rewriting lemma `inE` with the `mem_bindE` extensionality lemma for binders.

```
Definition eqbind_class := sub.

Identity Coercion sub_of_eqbind : eqbind_class >→ sub.

Coercion pred_of_eq_bind (σ : eqbind_class) : pred_class :=
  [eta mem_binding σ].

Canonical mem_bind_symtype := mkPredType mem_binding.

Lemma mem_bindE σ b : b ∈ σ = (σ b.1 == Some b.2).

Definition inE := (mem_bindE, inE).
```

One example of the usefulness of introducing this generic predicate is given in the next subsection, where we use it to define an ordering on substitutions.

Substitution Ordering In accordance with Definition 3.1.8, we say that a substitution σ_1 *extends* a substitution σ_2, i.e, $\sigma_1 \preceq \sigma_2$, if all variables bound by σ_1 appear in σ_2, bound to the same values. This is formalized in the `sub_st` boolean proposition and will be denoted as $\sigma_1 \subseteq \sigma_2$ in Coq proofs. Also, the SSReflect construct `(v, c)` $\in \sigma_2$

denotes σ_2 `v = Some c`. Note that it is possible to use the boolean universal quantifier (see Section 2.2), due to the finiteness of the ordinal type chosen to encode variables.

```
Definition sub_st σ1 σ2 :=
  [∀v : 'I_n, if σ1 v is Some c then (v, c) ∈ σ2 else true].
```

The more compact definition above is syntactic sugar provided by SSReflect for:

```
Definition sub_st σ1 σ2 :=
  [∀v : 'I_n, match σ1 v with
                | Some c ⇒ (v, c) ∈ σ2
                | None   ⇒ true
              end].
```

The characterizing reflection lemma `substP` below follows from the corresponding characterizing for the boolean universal quantifier:
`forallP : ∀(T : finType) (P : pred T), reflect (∀x : T, P x) [∀x, P x].`
Based on `substP`, we also establish the substitution extensionality lemma `substE`.

```
Lemma substP σ1 σ2 : reflect {subset σ1 ≤ σ2} (σ1 ⊆ σ2).
Lemma substE σ1 σ2 : reflect (∀c v, σ1 v = Some c → σ2 v = Some c)
                             (σ1 ⊆ σ2).
```

Substitutions are a key ingredient for the matching algorithm we present in Section 8.4.1. Indeed, the algorithm essentially consists of iteratively extending base substitutions with appropriate bindings for clause body atoms, in order to construct a grounding. This is then applied to the head to produce a new fact. Due to the crucial role substitutions play, it was necessary to build a small theory capturing their properties. For instance, we show these have a *partial order* structure.

```
(* reflexivity, antisymmetry and transitivity *)
Lemma substss σ : σ ⊆ σ .
Lemma subst_antitrans σ1 σ2 : σ1 ⊆ σ2 → σ2 ⊆ σ1 → σ1 = σ2 .
Lemma subst_trans σ1 σ2 σ3 : σ1 ⊆ σ2 → σ2 ⊆ σ3 → σ1 ⊆ σ3.
```

Extending a substitution is realized by the `add` function, for which we prove soundness (`sub_add`) and extensionality (`addE`). Also we establish in the `add_add` lemma that adding a new binding for the same variable shadows the previous one.

```
Definition add σ v c :=
  [ffun u ⇒ if u == v then Some c else σ u].

Lemma sub_add σ v c : mem_free s v → σ ⊆ (add σ v c).
```

```
Lemma addE σ v c : (add σ v c) v = Some c.
Lemma add_add σ v c e : (add (add σ v e) v c) = add σ v c.
```

A very important aspect, as mentioned above, is establishing groundedness of substituted constructs, under certain restrictions. As such, it is needed to reason about the *domain* of substitutions (`dom`), in particular in the context of exploiting the safety condition. The base lemma is `sub_dom_grt`, according to which a (variable) term is grounded by any substitution whose domain subsumes it. Similarly, the `sub_dom_gra` and `sub_dom_ga` reflection lemmas establish, for any (raw) atom and substitution, an equivalence between the substitution being grounding and the (raw) atom variables belonging to the substitution domain. The `sub_dom_gtl` lemma proves the analogous result for lists of atoms.

```
Definition dom σ := [set v : 'I_n | s v].

Lemma sub_dom_grt t σ :
  term_vars t ⊆ dom σ ↔ ∃c, sterm σ t = Val c.

Lemma sub_dom_gra ra σ :
  reflect (∃gra, sraw_atom ra σ = to_raw_atom gra)
          (raw_atom_vars ra ⊆ dom σ).

Lemma sub_dom_ga a σ :
  reflect (∃ga, satom a σ = to_atom ga)
          (atom_vars a ⊆ dom σ).

Lemma sub_dom_gtl tl σ :
  reflect (∃gtl, stail tl σ = [seq to_atom ga | ga ← gtl])
          (tail_vars tl ⊆ dom σ).
```

Based on these, we prove the following substitution extension lemmas:

```
Lemma sterm_sub t σ1 σ2 d :
  σ1 ⊆ σ1 → sterm σ1 t = Val d → sterm σ2 t = Val d.

Lemma satom_sub a σ1 σ2 ga :
  σ1 ⊆ σ1 → satom a σ1 = to_atom ga → satom a σ2 = to_atom ga.
```

As shown in Section 8.5.8, the main substitution ordering lemmas were instrumental to proving the soundness of body matching and of the clause consequence operator.

8.3.3. Relating Groundings and Substitutions

Grounding substitutions can be modeled *directly* and non-dependently, using a padding default constant `def`. As such, given a substitution σ, we encode its grounding application to an arbitrary term `t`, to a raw atom `ra` and to an atom `a`, in the `gr_term_def`, `gr_raw_atom_def` and `gr_atom_def` definitions below. For the former, we use the `odflt` function from `ssrfun` and case on `t`: if it is a constant, we do not modify it and, if it is a variable, i.e, `Var v`, we return `odflt def (σ v)`. That is to say, we either return the substituted variable `σ v`, if it exists, or `def`, otherwise. The corresponding soundness lemma `gr_term_defP` is provable by casing.

The grounding of a raw atom `ra` with σ, i.e, `gr_raw_atom_def σ ra`, is built wrapping in the `RawGAtom` constructor the symbol of the raw atom, together with the grounding application of σ to all of the raw atom arguments. Finally, in `gr_atom_def`, to express the grounding of an atom `a` with σ, it suffices to wrap in the `GAtom` constructor the well-formedness proof `gr_atom_def_proof σ a` for the grounding of its underlying raw atom. The latter is a direct consequence of the raw atom well-formedness and of the `size_map` lemma, stating list size is invariant to function mapping. The soundness lemma `gr_atom_defP` below is provable by double induction on the argument lists of `a` and `ga` , using `gr_term_defP`.

```
Variable def : constant.

Definition gr_term_def σ t : constant :=
  match t with
  | Val c ⇒ c
  | Var v ⇒ odflt def (σ v)
  end.

Lemma gr_term_defP c t σ :
  sterm σ t = Val c → gr_term_def σ t = c.

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

Lemma gr_atom_def_proof σ a : wf_gatom (gr_raw_atom_def σ a).
Proof. by case: a ⇒ ra pf; rewrite /wf_gatom size_map. Qed.

Definition gr_atom_def σ a : gatom := GAtom (gr_atom_def_proof σ a).

Lemma gr_atom_defP a σ ga :
  satom a s = to_atom ga → gr_atom_def σ a = ga.
```

Using `def`, we can also *indirectly* lift a substitution `σ : sub` to a grounding $\sigma_{\texttt{def}}$, by defin-

ing a transformation function, i.e, `to_gr`, that provides the respective finitely-supported function. As discussed in Section 8.1.4, this is equivalent to building the tabulation of the latter, which has the type `n.-tuple constant`. Consequently, for each variable `v` in the finite (ordinal) domain `'I_n` of σ, we inspect whether σ binds `v` to a constant `c`. If so, we return `c` and, if not, we pad the tuple corresponding to the finitely-supported grounding with `def`. Conversely, a grounding ν `: gr` can be transformed into a substitution $\hat{\nu}$, through the `to_sub` function. For each variable `v` in the finite (ordinal) domain `'I_n` of ν, this converts the application of the grounding ν `v` into an option type.

```
Definition to_gr (σ : sub) : gr :=
  [ffun v ⇒ if σ v is Some c then c else def].

Definition to_sub (ν : gr) : sub := [ffun v ⇒ Some (ν v)].
```

8.3.4. Program Semantics

We present the main ingredients for capturing the semantics of a standard Datalog program: the type of interpretations, i.e, `interp`, and the definition of program satisfiability with respect to a model, i.e, `prog_true`.

Program Interpretation An interpretation `i`, for a program `p`, is a *finite set* of ground atoms (facts). Note that `gatom` is a assumed to share the signature of `p`; that is to say, it is built over the same types of symbols and constants.

```
Notation interp := {set gatom}.
```

As seen in Section 5, an interpretation function assigns concrete meaning to the constants and symbols of a program. A clause `cl` of the form $H \leftarrow B_1, \ldots, B_m$ can have *multiple interpretations*, rendering it either true or false. A *particular* interpretation `i` is a model of `cl`, if, for all groundings ν, when the clause's body groundings belong to `i`, i.e $\{\nu B_1, \ldots, \nu B_m\} \subseteq$ `i`, then its head grounding also belongs to `i`, i.e, $\nu H \in$ `i`.

This textbook definition of logical consequence is encoded in the `cl_true` proposition. We express the fact that an interpretation `i` satisfies a clause `cl`, by stating that, for all valuations ν, `i` satisfies the corresponding ground clause `gr_cl` ν `cl`. To this end, we separately define ground clause interpretations, via the boolean proposition `gcl_true`. This ensures the same valuation *homogeneously* instantiates all ground body atoms.

```
Definition gcl_true gcl i :=
  all (mem i) (body_gcl gcl) ==> (head_gcl gcl ∈ i).
```

An interpretation `i` is a model for a program `p`, if `i` satisfies all clauses `cl` in `p`. Hence, we verify that, for all valuations ν, the interpretations of ground clauses, obtained instantiating `cl` with ν, are true.

```
Definition prog_true p i  :=
  ∀ν : gr, all (fun cl ⇒ gcl_true (gr_cl ν cl) i) p.
```

8.4. Bottom-up Evaluation

The key in formalizing the bottom-up evaluation of a standard DATALOG program is its underlying matching algorithm, presented in Section 8.4.1. Based on this, in Section 8.4.2, we define the one-step forward chain, implementing logical consequence.

8.4.1. Matching Algorithm

The matching algorithm consists of *incrementally* constructing substitutions that homogeneously instantiate all atoms in a given clause body to facts in the candidate program model. As such, all of our matching functions will also take a seed substitution parameter σ. This represents the previous intermediate substitution to be extended. The main `match_body` procedure is built up through the term matching function `match_term` and the atom matching function `match_atom`, presented as follows.

Term Matching The `match_term` function, matching a term t to a constant d under a substitution σ, will either: 1) return the *input substitution* intact, if the term t or the substituted term $\sigma\ t$ already equals d, 2) return an *extended substitution* for σ, if t is a variable that has not been previously bound in σ, or 3) *fail*, if t or the substituted term $\sigma\ t$ are different from d.

```
Definition match_term d t σ : option sub :=
  match t with
   | Val e ⇒ if d == e then Some σ else None
   | Var v ⇒ if σ v is Some e
             then (if d == e then Some σ else None)
             else Some (add σ v d)
  end.
```

Atom Matching We define atom matching functions `match_atom` and `match_atom_all` that, respectively, return substitutions (sets of substitutions) instantiating a given atom to a ground atom (set of ground atoms). To compute the substitution matching a raw-atom `ra` to a ground one `rga`, we first have to check that the symbols and argument sizes of the atoms agree. If such, we cumulatively enrich the initial accumulator value σ,

by iterating term matching over `zip arg2 arg1`, i.e, the itemwise pairing of the atoms' terms. Since term matching can fail, we wrap the function with an option binder extracting the corresponding variable assignations, if they exists. Hence, `match_raw_atom` is, in effect, a monadic option fold that either fails or produces substitution extensions of σ.

```
Definition match_raw_atom rga ra σ : 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 σ) (zip arg2 arg1)
      else None
  end.
```

Atom matching can be defined as equal to the matching of the underlying raw atom matching, due to the coercion to `raw_atom`.

```
Definition match_atom σ a ga := match_raw_atom σ a ga.
```

Next, we compute the substitutions that can match an atom `a` to a fact in an interpretation `i`. This is formalized as the set containing substitutions σ that belong to the set gathering all substitutions matching `a` to ground atoms `ga` in `i`.

```
Definition match_atom_all i a σ :=
  [set σ' | Some σ' ∈ [set match_atom ga a σ | ga ∈ i]].
```

It is interesting to note that, while the previous matching functions `match_term` and `match_atom` were written as Gallina algorithms, we were able to abstract to a more declarative level in defining `match_atom_all`. Indeed, even though `match_atom_all` implements an *algorithm*, its essence is set-theoretic, i.e:

$$\{\sigma' \mid \sigma' \in \{\texttt{match\_atom ga a } \sigma \mid \texttt{ga} \in \texttt{i}\}\}$$

The function is at the basis of expressing forward chain (seen Section 8.4.2) and, ultimately, at the basis of fixpoint evaluation. Consequently, by propagating its implementation, we manage to "reduce" proofs about the soundness and completeness of underlying algorithms, to mathematical proofs belonging to set theory. As such, it was particularly convenient that we could rely on properties established in the `finset` library.

Body Matching Having all the required ingredients, we proceed to define the matching for clause bodies `match_body`, crucial to the inferrence of new facts (see Section 8.3.4).

We capture the fact that this operation returns a collection of substitution computations, by modeling it with `foldS`, a monadic fold for the set monad. The operator iteratively

composes the result of applying a function `f`, seeded with an initial value σ_0, to all the elements of a list `l`, while flattening intermediate outputs.

```
Fixpoint foldS {A : Type} {B : finType}
         (f : A → B → {set B}) (σ₀ : {set B}) (l : seq A) :=
  if l is [:: x & l] then bindS σ₀ (fun y ⇒ foldS f (f x y) l)
  else σ₀.
```

We implement the flattening via `bindS`, given below. We modeled it using the `cover` operator, from the `finset` library. For a given set of sets `S`, this takes the union of its elements, i.e, `cover S := \bigcup_(x in S) x` [4]. In our setting, over each of these elements, we can additionally map an arbitrary function `f`.

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

The function `match_body` extends an initial set of substitutions `ss0` with the results of matching all atoms in the `tl` body of a clause, to an interpretation `i`. The latter are built using `match_atom_all` and stepwise and uniformly extending substitutions matching each body atom to `i`.

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

As we will see in Section 8.5.1, in order to prove the soundness of `match_body`, a key result is the following characterization lemma for `bindS`.

```
Lemma bindP {A B : finType} (S : {set A})
            (f : A → {set B}) (σ : B) :
  reflect (∃ θ, θ ∈ S & σ ∈ f θ) (σ ∈ bindS S f).
Proof.
by rewrite/bindS cover_imset; exact:(iffP bigcupP); case⇒ s ??; ∃s.
Qed.
```

Regarding the proof for `bindP`, we have that the `cover_imset` lemma, ensures: `cover [set f x | x in S]` $= \bigcup_{x \in S}$ `f x`. Finally, from the characteristic lemma `bigcupP` for big union, it holds that $\sigma \in \bigcup_{x \in S}$ `f x` is equivalent to `∃ θ, θ ∈ S & σ ∈ f θ`.

8.4.2. Building the Forward Chain Operator

We mechanize the forward chain of a program as the iteration of a `fwd_chain` function that collects the results of applying a consequence operator (as described in Section 5)

[4] In mathematical notation, this is equivalent to `cover S` $:= \bigcup_{x \in S}$ `x`

to the program's clauses.

Immediate Consequence Operator Given an interpretation `i`, we model the immediate consequences of a clause `cl` as the set of *new* facts that can be inferred from `cl`, by matching its body to `i`. One such fact `gr_atom_def def` σ `(head_cl cl)` is constructed by instantiating the clause head with the grounding corresponding to the matching substitution σ. Note that `emptysub` encodes the empty substitution.

```
Definition emptysub : sub := [ffun _ ⇒ None].

Definition cons_clause def cl i :=
  [set gr_atom_def def σ (head_cl cl) |
       σ ∈ match_body i (body_cl cl) [set emptysub]].
```

One-Step Forward Chain One iteration of the forward chain algorithm computes the set of all consequences that can be inferred from a program `p` and an interpretation `i`. This amounts to taking the union of `i` and of all the consequences of the program's clauses. The formalization naturally mirrors the mathematical expression $\texttt{i} \cup \bigcup_{\texttt{cl} \in \texttt{p}} \texttt{cons\_clause def i cl}$, due to the use of ssreflect's big operator, i.e, `\bigcup`.

```
Definition fwd_chain def p i :=
  i ∪ \bigcup_(cl ← p) cons_clause def i cl.
```

8.5. Bottom-Up Evaluation Properties

This section details the fundamental results characterizing the matching algorithm (Section 8.5.1) and the one-step forward chain (Section 8.5.2).

8.5.1. Matching Characterization

In the following, we establish *soundness* and *completeness* of our matching algorithm.

I. Soundness of Matching

We present the *soundness* proofs for the algorithms matching terms, atoms and clause bodies, presented in Section 8.4.1. That is, we show that the output substitutions are indeed a solution for the respective matching problems.

Term Matching Soundness

Lemma 8.5.1. *Let* t *be a term,* d *a constant and* σ*, an arbitrary substitution. If* `match_term` *outputs a substitution* θ*, extending* σ *with the matching of* t *to* d*, then* θ *is indeed a solution, i.e the instantiation of* t *with* θ *equals* d*.*

```
Lemma match_term_sound d t σ θ :
  match_term d t σ = Some θ → sterm θ t = Val d.
```

Proof. The proof follows by case analysis on `t`.

Case 1 If `t` is a constant `c`, matching returns a substitution θ, only if `c` equals `d`, in which case $\theta = \sigma$. Substitution application then trivially satisfies the conclusion.

Case 2 If `t` is a variable `v`, we analyze the following cases: $\sigma\ v = d$ and $\sigma\ v = \bot$, which lead to $\theta = \sigma$ and, respectively, $\theta = \sigma, [d/v]$. Either way, $\theta\ v = d$.

□

Atom Matching Soundness

Lemma 8.5.2. *Let* a *be an atom,* ga *a ground atom and* σ*, an arbitrary substitution. If* `match_atom` *outputs a substitution* θ*, extending* σ *with the matching of* a *to* ga*, then* θ *is indeed a solution, i.e the instantiation of* a *with* θ *equals* ga*.*

```
Lemma match_atom_sound a ga σ θ :
  match_atom σ a ga = Some θ → satom θ a = to_atom ga.
```

Proof. We destruct the atom `a` and the ground atom `ga` into their raw atom and, respectively, raw ground atom counterparts, together with their well-formedness proofs. For the atom matching algorithm to output a substitution θ, the corresponding symbols and argument sizes have to agree. To prove the ground atom obtained instantiating `a` with θ equals `ga`, we proceed by induction on the atom's argument list. The base case trivially holds, since θ `a` and `ga` have equal symbols and empty argument lists. In the step case, the arguments of `a` and `ga` are `t :: tl_a` and `c :: tl_g`. By congruence and application of the induction hypothesis, the proof boils down to showing `sterm` θ `t = Val c`. To relate the matching of head terms, the intermediate matching of tail terms and the final matching, we prove:

```
Lemma foldl_0 gl l σ θ :
  foldl (fun acc p ⇒ obind [eta match_term p.1 p.2] acc)
    σ (zip gl l) = Some θ → ∃ η, σ = Some η & η ⊆ θ.
```

Hence, there exists a substitution η, such that `match_term c t σ = Some η` and $\eta \subseteq \theta$, i.e, the matching `a` to `ga` extends η with intermediate bindings, from the pairwise term matching of `tl_a` and `tl_g`. As a result of the substitution extension lemma for terms, we can restate our goal as `sterm η t = Val c`. This holds, according to the soundness of term matching (Lemma 8.5.1). □

Regarding atom matching, another useful property - used in the next soundness lemma - is `match_atom_sub`. This establishes an ordering between the input substitution of `match_atom` and its output. Its proof is by casing and direct application of `foldl_0`.

```
Lemma match_atom_sub σ1 σ2 a ga :
  match_atom σ1 a ga = Some σ2 → σ1 ⊆ σ2.
```

Body Matching Soundness

We begin by introducing two auxiliary lemmas concerning the `match_atom_all` function, which underlies the definition of `match_body`.

First, we establish its soundness, expressed by `match_atomsP` below. The proof is a corrolary of the `imsetP` lemma, from `finset`. This essentially establishes that, for finite sets A and B, for an arbitrary function $f : A \to B$ and an element $y \in B$, the fact that $y \in fA$ is equivalent to the existence of an element $x \in A$, such that $y = fx$.

```
Lemma match_atomsP a i θ σ :
  reflect (∃ ga, ga ∈ i & Some θ = match_atom σ a ga)
          (σ ∈ match_atom_all i a σ).
```

Second, we show that we can view a substitution σ, matching a list of atoms `a :: l` to interpretation `i`, as the extension of a substitution matching `a` to `i`, with bindings matching atoms in `l` to `i`. This extended substitution is the result of seeding `match_body` with the output of `match_atom_all`.

```
Lemma match_body_cons a l i σ ss0 :
  reflect
    (∃ θ, θ ∈ ss0 & σ ∈ match_body i l (match_atom_all i a θ))
    (σ ∈ match_body i (a :: l) ss0).
```

Indeed, unfolding `match_body i (a :: l) ss0`, we obtain:

```
foldS (match_atom_all i) ss0 (a :: l).
```

This, in turn, corresponds to:

```
bindS ss0 (fun y ⇒ foldS (match_atom_all i) (match_atom_all i a y) l),
```

which is the same as:

```
bindS ss0 (fun y ⇒ match_body i l (match_atom_all i a y)).
```

It then follows $\sigma \in$ `match_body i (a :: l) ss0` is equivalent to:

```
σ ∈ bindS ss0 (fun y ⇒ match_body i l (match_atom_all i a y)).
```

The fact that `∃ θ, θ ∈ ss0 & σ ∈ match_body i l (match_atom_all i a θ)` is a corollary of the `bindP` monadic bind reflection lemma from Section 8.4.1.

Next, we prove a similar result to `match_atom_sub`, extended to body atom matching and sets of input substitutions. This is expressed in the `match_body_sub` lemma that follows. This states that, given an interpretation `i`, if σ is in the result of the extending substitutions from a set `ss0` with bindings matching an atom list `tl` to `i`, then there is a substitution θ in `ss0`, such that $\theta \preceq \sigma$.

```
Lemma match_body_sub tl i σ ss0 :
  σ ∈ match_body i tl ss0 → ∃ θ, θ ∈ ss0 & θ ⊆ σ.
```

The proof is by induction on `tl`, using `match_atom_sub` and the transitivity of substitution ordering `subst_trans` (see Section 8.3.2).

We continue with the proof for the soundness of body matching, stated as follows.

Lemma 8.5.3. *If a substitution σ is in the output of* `match_body`*, extending a given substitution set* ss0 *with matchings of a clause body* tl *to an interpretation* i*, then there exists a ground body* gtl *such that*

- gtl *is the instantiation of* tl *with* θ
- *all of the atoms in* gtl *belong to* i

```
Lemma match_body_sound tl i σ ss0 :
  σ ∈ match_body i tl ss0 →
      ∃ gtl, stail tl σ = [seq to_atom ga | ga ← gtl]
             & all (mem i) gtl.
```

Proof. The proof is by structural induction on the clause body atom list `tl`.

Base Case `tl` ≡ `[ ]`. Trivial.

Induction Case `tl` ≡ `l`.

Induction Hypothesis `tl` ≡ `a :: l`. We know:

$$\forall \texttt{ss0}\ \forall \sigma,\ \sigma \in \texttt{match\_body i l ss0} \rightarrow$$
$$\exists\ \texttt{gtl, stail tl}\ \sigma = \texttt{[seq to\_atom ga | ga} \leftarrow \texttt{gtl] \& all (mem i) gtl.}$$

Assume a substitution set `ss0` and a substitution σ, where

$$\sigma \in \texttt{match\_body i (a :: l) ss0.}$$

From `match_body_cons`, it holds that

$$\exists\ \theta,\ \theta \in \texttt{ss0 \&}\ \sigma \in \texttt{match\_body i l (match\_atom\_all i a}\ \theta\texttt{).}$$

Instantiating the induction hypothesis with `(match_atom_all i a` θ`)` as `ss0`, with σ and with $\sigma \in$ `match_body i l (match_atom_all i a` θ`)`, it follows that:
$\exists$ `gtl', stail l` $\sigma =$ `[seq to_atom ga | ga` $\leftarrow$ `gtl'] & all (mem i) gtl'`

From $\sigma \in$ `match_body i l (match_atom_all i a` θ`)`, by `match_body_sub`:

$$\exists\ \theta',\ \theta' \in \texttt{match\_atom\_all i a}\ \theta\ \texttt{\&}\ \theta' \subseteq \sigma.$$

Applying `match_atomsP` to $\theta' \in$ `match_atom_all i a` θ, we obtain:

$$\exists\ \texttt{ga', ga'} \in \texttt{i \&}\ \theta' = \texttt{match\_atom i a}\ \theta$$

Atom matching soundness ensures `satom` θ' `a = to_atom ga'`. From this and $\theta' \subseteq \sigma$, we derive, by `satom_sub`, that `satom a` σ `= to_atom ga'`.

We take `ga' :: gtl'` as the witness ground atom list `gtl`.
`satom a` σ `= to_atom ga'` and `stail l` σ `= [seq to_atom ga | ga` $\leftarrow$ `gtl']` lead to `stail (a :: l)` σ `= [seq to_atom ga | ga` $\leftarrow$ `(ga' :: gtl')]`.
Finally, `ga'` $\in$ `i` and `all (mem i) gtl'` imply `all (mem i) (ga' :: gtl')`.

□

II. Completeness of Matching

We detail the *completeness* proofs for the algorithms matching terms, atoms and clause bodies, presented in Section 8.4.1. That is, we show these output *all* possible solutions for their respective matching problems. In fact, these algorithms produce the *best* solutions, which is something we account for in our theorem statements, as follows.

Term Matching Completeness

Lemma 8.5.4. *Let* t *be a term,* d *a constant and* σ *a substitution that is a solution for matching* t *against* d. *The* `match_term` *algorithm seeded with* σ'*, an arbitrary restriction of* σ*, outputs a better matching solution than* σ*, i.e a substitution* θ *that is smaller or equal to* σ*.*

```
Lemma match_term_complete d t σ σ' :
  σ' ⊆ σ → sterm σ t = Val d →
  ∃ θ, match_term d t σ' = Some θ & θ ⊆ σ.
```

Proof. The proof follows by case analysis on `t`.

Case 1 If `t` is a constant `c`, the instantiation of `t` equals `d`, only if `c` equals `d`. Hence, a trivial solution to the matching algorithm is the input substitution σ'.

Case 2 If `t` is a variable `v`, we know that it is instantiated by σ and proceed to inspecting whether of not it is also instantiated by σ'. If so, both σ `v` and σ' `v` are equal and the matching algorithm returns σ'. If not, the matching algorithm returns θ, the extension of σ' with the corresponding binding for `v`, i.e, θ = `add` σ' `v d`, where $\sigma' \subseteq \sigma$ and σ `v` = `d`.

□

Atom Matching Completeness

Lemma 8.5.5. *Let* a *be an atom,* ga *a ground atom and* σ *a substitution that is a solution for matching* a *against* ga*. The* `match_atom` *algorithm seeded with* σ'*, an arbitrary restriction of* σ*, outputs a better matching solution than* σ*, i.e a substitution* θ *that is smaller or equal* σ*.*

```
Lemma match_atom_complete ga a σ σ' :
  σ' ⊆ σ → satom σ a = to_atom ga →
  ∃ θ, match_atom σ' a ga = Some θ & θ ⊆ σ.
```

Proof. Since instantiating `a` with σ equals `ga`, the symbols and argument sizes of `a` and `ga` have to agree. Hence, the corresponding matching algorithm will produce an output. We build the witness substitution by simultaneous induction on `arg_a` and `arg_ga`, the argument lists for `a` and `ga`. For the base case, we take the witness to be σ', which trivially satisfies the required conditions. For the step case, we distinguish between the head term `t` of `arg_a` and the head constant `c` of `arg_ga`, on the one hand, and the tail arguments `arg_a'` and `arg_ga'`, on the other. From the hypothesis, we infer that `sterm` σ `t` = `c` and that the mapping of σ over `arg_a'` equals `arg_ga'`. Applying the term matching completeness lemma for `t` and `c`, we extract the substitution θ', such that `match_term c t` σ' = `Some` θ'. Finally, θ is obtained from the induction hypothesis, by passing θ' as a seed. □

Body Matching Completeness

Lemma 8.5.6. *Let* cl *be a clause,* i *an interpretation and* ν*, a valuation compatible with any substitution* σ *in the accumulated substitution set* `ss0`*. If* ν *makes the body of* cl *true in* i*, then the* `match_body` *algorithm outputs a compatible substitution* θ *that is smaller or equal to* ν*.*

```
Lemma match_body_complete σ ss0 i cl ν :
  σ ∈ ss0 → σ ⊆ (to_sub ν) →
  all (mem i) (body_gcl (gr_cl ν cl)) →
  ∃ θ, θ ∈ match_body (body_cl cl) i ss0 & θ ⊆ (to_sub ν).
```

Proof. Let us fix the grounding ν, an interpretation i and the clause $\text{cl} \equiv H \leftarrow B_1, \ldots, B_n$. The proof is by structural induction on the clause body list.

Base Case $\text{cl} \equiv H \leftarrow$.

Since `match_body` $[\,]$ i $\text{ss}_0 = \text{ss}_0$, we trivially take σ as the witness substitution θ.

Induction Case $\text{cl} \equiv H \leftarrow B_0, B_1, \ldots, B_n$.

Induction Hypothesis $\text{cl} \equiv H \leftarrow B_1, \ldots, B_n$. We know that:

$$\begin{aligned} \forall \text{ss}_0\ \forall \sigma,\ \sigma \in \text{ss}_0\ \wedge\ \sigma \preceq \hat{\nu}\ \wedge\ \{\nu B_1, \ldots, \nu B_n\} \subseteq \text{i} \Rightarrow \\ \exists \theta,\ \theta \in \texttt{match\_body}\ [B_1, \ldots, B_n]\ \text{i}\ \text{ss}_0\ \wedge\ \theta \preceq \hat{\nu}. \end{aligned} \tag{8.1}$$

Recall from Section 8.3.3, that we denote the coercion of a valuation (grounding) ν to a substitution as $\hat{\nu}$. Assume a substitution set ss_0 and a substitution σ, where $\sigma \in \text{ss}_0$. Also, assume a ground substitution η, with $\sigma \preceq \hat{\eta}$ and $\{\eta B_0, \ldots, \eta B_n\} \subseteq$ i. Since $\eta B_0 \in$ i:

$$\exists \text{ga},\ \text{ga} \in \text{i}\ \wedge\ \eta B_0 = \text{ga} \tag{8.2}$$

From $\sigma \preceq \hat{\eta}$ and $\eta B_0 = \text{ga}$, applying the `match_atom_complete` lemma, we obtain:

$$\exists \eta',\ \texttt{match\_atom}\ \sigma\ B_0\ \text{ga} = \eta'\ \wedge\ \eta' \preceq \hat{\eta} \tag{8.3}$$

Using the reflection lemma `match_atomsP` on 8.2 and `match_atom` $\sigma\ B_0\ ga = \eta'$:

$$\eta' \in \texttt{match\_atom\_all}\ \text{i}\ B_0\ \sigma \tag{8.4}$$

Instantiating the induction hypothesis with (`match_atom_all` i $B_0\ \sigma$) as ss_0 and η' as σ, together with $\eta' \preceq \eta$ (from 8.3) and $\{\eta B_1, \ldots, \eta B_n\} \subseteq$ i, it follows that:

$$\exists \theta,\ \theta \in \texttt{match\_body}\ \text{i}\ [B_1, \ldots, B_n]\ (\texttt{match\_atom\_all}\ \text{i}\ B_0\ \sigma) \wedge \theta \preceq \eta \tag{8.5}$$

To conclude the proof, from 8.5 and $\sigma \in \text{ss}_0$, applying `match_body_cons`:

$$\theta \in \texttt{match\_body}\ \text{i}\ [B_0, \ldots, B_n]\ \text{ss}_0 \tag{8.6}$$

□

Note that the previous `match_body_complete` lemma is a generalized, "technical" one. From it, we can obtain, as a corollary, the standard version:

```
Lemma match_body_complete_gen σ i cl ν :
  all (mem i) (body_gcl (gr_cl ν cl)) →
  ∃θ, θ ∈ match_body (body_cl cl) i [set emptysub] & θ ⊆ (to_sub ν).
```

8.5.2. Forward-Chain Characterization

In the following, we establish the *stability* and *soundness* properties of logical consequence inference and, based on these, the analogous properties of one-step forward chain.

Immediate Consequence Operator Characterization

The main properties of the `cons_clause` operator are *stability* and *soundness*, whose proofs are given below.

Theorem 8.5.7 (Stability of the Immediate Consequence Operator)**.** *Let* cl *be a positive clause and* i *a* model *for* cl*. The application of the consequence operator on* cl*, given* i*, is* stable*, i.e, the operator derives no new facts from* cl*.*

This is encoded as:

```
Lemma cons_cl_stable def cl i :
  cl_true cl i → cons_clause def cl i ⊆ i.
```

Proof. Let $\mathrm{cl} \equiv H \leftarrow B_1, \ldots, B_n$. Proving the *stability* property amounts to showing:

$$\{\sigma_{\mathrm{def}}\ H \mid \sigma \in \texttt{match\_body}\ \mathrm{i}\ [B_1, \ldots, B_n]\ \texttt{[set emptysub]}\} \subseteq \mathrm{i}.$$

We apply the `sub_imset_pre` lemma, according to which, for any finite sets A and B and function $f : A \to B$, it holds that $\{f(x) \mid x \in A\} \subseteq B \Leftrightarrow A \subseteq f^{-1}(B)$. Hence, our goal is transformed into `match_body` i $[B_1, \ldots, B_n]$ `[set emptysub]` $\subseteq \{\sigma \mid \sigma_{\mathrm{def}}\ H \in \mathrm{i}\}$. Let $\sigma \in$ `match_body` i $[B_1, \ldots, B_n]$ `[set emptysub]`. We aim to infer $\sigma_{\mathrm{def}}\ H \in \mathrm{i}$.

From the *body matching soundness*, we know there exists a list of ground atoms $[\bar{B}_1, \ldots, \bar{B}_n]$, where $\{\bar{B}_1, \ldots, \bar{B}_n\} \subseteq \mathrm{i}$ and $[\sigma B_1, \ldots, \sigma B_n] = [\ulcorner \bar{B}_1 \urcorner, \ldots, \ulcorner \bar{B}_n \urcorner]$. The latter is equivalent to $[\sigma_{\mathrm{def}}\ B_1, \ldots, \sigma_{\mathrm{def}}\ B_n] = [\bar{B}_1, \ldots, \bar{B}_n]$. Hence, $\{\sigma_{\mathrm{def}}\ B_1, \ldots, \sigma_{\mathrm{def}}\ B_n\} \subseteq \mathrm{i}$.

From the hypothesis, we know $\mathrm{i} \models \mathrm{cl}$, which means that, for all ground substitutions η, if $\{\eta B_1, \ldots, \eta B_n\} \subseteq \mathrm{i}$, then $\eta H \in \mathrm{i}$. Taking η to be σ_{def}, together with $\{\sigma_{\mathrm{def}}\ B_1, \ldots, \sigma_{\mathrm{def}} B_n\} \subseteq \mathrm{i}$, concludes the proof. □

Theorem 8.5.8 (Soundness of Clause Consequence)**.** *Let* cl *be a* safe *positive clause and* i*, an arbitrary interpretation. If the application of the consequence operator on* cl *given* i *is* stable*, i.e, the operator derives no new facts from* cl*, then* i *is a* model *for* cl*.*

This is encoded as:

```
Lemma cons_cl_sound def cl i :
  safe_cl cl → cons_clause def cl i ⊆ i → cl_true cl i .
```

Proof. Let $\mathrm{cl} \equiv H \leftarrow B_1, \ldots, B_n$. Unfolding the definition corresponding to consequence operator application, the *stability* hypothesis becomes:

$$\{\sigma_{\mathrm{def}}\ H \mid \sigma \in \texttt{match\_body}\ \mathrm{i}\ [B_1, \ldots, B_n]\ \texttt{[set emptysub]}\} \subseteq \mathrm{i}.$$

Proving i is a model for cl means showing that, for any ground substitution η, if $\{\eta B_1, \ldots, \eta B_n\} \subseteq \mathrm{i}$, then $\eta H \in \mathrm{i}$. Given the above condition, it suffices to establish:

$$\eta H \in \{\sigma_{\mathrm{def}}\ H \mid \sigma \in \texttt{match\_body}\ \mathrm{i}\ [B_1, \ldots, B_n]\ \texttt{[set emptysub]}\}$$

i.e, that $\exists \theta,\ \theta \in$ `match_body` i $[B_1, \ldots, B_n]$ `[set emptysub]`, such that $\eta H = \theta_{\mathrm{def}}\ H$.

From the corollary of *body matching completeness* and $\{\eta B_1, \ldots, \eta B_n\} \subseteq \mathrm{i}$:

$$\exists \theta,\ \theta \in \texttt{match\_body}\ \mathrm{i}\ [B_1, \ldots, B_n]\ \texttt{[set emptysub]} \text{ and } \theta \preceq \eta.$$

Hence, it only remains to show $\eta H = \theta_{\mathrm{def}}\ H$, i.e $\hat{\eta}_{\mathrm{def}}\ H = \theta_{\mathrm{def}}\ H$.

To this end, we first prove there exists a ground atom $\bar{H}$, such that $\theta\, H = \ulcorner \bar{H} \urcorner$, which, by the `sub_dom_ga` reflection lemma (see Section 8.3.2) is equivalent to $\mathrm{Var}\, H \subseteq \mathrm{dom}\ \theta$. In turn, from transitivity and the *safety* condition $\mathrm{Var}\, H \subseteq \mathrm{Var}\, [B_1, \ldots, B_n]$, this reduces to proving $\mathrm{Var}\, [B_1, \ldots, B_n] \subseteq \mathrm{dom}\ \theta$. From $\theta \in$ `match_body` i $[B_1, \ldots, B_n]$ `[set emptysub]` and *body matching soundness*, it follows there exists a list of ground atoms $[\bar{B}_1, \ldots, \bar{B}_n]$, such that $[\theta B_1, \ldots, \theta B_n] = [\ulcorner \bar{B}_1 \urcorner, \ldots, \ulcorner \bar{B}_n \urcorner]$. By the `sub_dom_gtl` reflection lemma, it holds there exists a ground atom $\bar{H}$, such that $\theta\, H = \ulcorner \bar{H} \urcorner$.

Applying the atom grounding reflection lemma `gr_atom_def` to the above: $\theta_{\mathrm{def}}\, H = \bar{H}$. Also, since $\theta \preceq \eta$ and $\theta H = \ulcorner \bar{H} \urcorner$, from `satom_sub`, $\hat{\eta}\, H = \ulcorner \bar{H} \urcorner$, i.e, $\hat{\eta}_{\mathrm{def}}\, H = \bar{H}$. Consequently, $\hat{\eta}_{\mathrm{def}}\, H = \theta_{\mathrm{def}}\, H$. □

One-Step Forward Chain Characterization

The main properties of the one-step forward chain are *stability* and *soundness*. Based on these, we can state its characterization `fwd_chainP`, as a reflection lemma.

Theorem 8.5.9 (Stability of One-Step Forward Chain)**.** *Let* p *be a positive program. Any interpretation* i *that is a* model *of* p *is a* fixpoint *of one iteration of forward chain.*

This is encoded as:

```
Lemma fwd_chain_stable def p i :
  prog_true p i → fwd_chain def p i = i.
Proof.
move⇒ p_true; apply/setUidPl/bigcups_seqP⇒ cl h_in _.
by apply/cons_cl_stable⇒ v; have/allP := p_true v; exact.
Qed.
```

Proof. Unfolding the definition of a forward chain step, the *fixpoint* property becomes:

$$\mathrm{i} \cup \bigcup_{\mathrm{cl} \in \mathrm{p}} \texttt{cons\_clause}\ \mathrm{def}\ \mathrm{cl}\ \mathrm{i} = \mathrm{i}.$$

Applying the `setUidPl` reflection lemma, this is equivalent to:

$$\bigcup_{\mathrm{cl} \in \mathrm{p}} \texttt{cons\_clause}\ \mathrm{def}\ \mathrm{cl}\ \mathrm{i} \subseteq \mathrm{i}.$$

which, in turn, by the `bigcups_seqP` reflection lemma, becomes:

$$\forall \mathrm{cl},\ \mathrm{cl} \in \mathrm{p}, \texttt{cons\_clause}\ \mathrm{def}\ \mathrm{cl}\ \mathrm{i} \subseteq \mathrm{i}.$$

Since $\mathrm{i} \models \mathrm{p}$, it follows that, for any clause cl, where $\mathrm{cl} \in \mathrm{p}$, $\mathrm{i} \models \mathrm{cl}$. Hence, the conclusion follows directly from Theorem 8.5.7. □

Theorem 8.5.10 (Soundness of One-Step Forward Chain)**.** *Let* p *be a* safe *positive program and* i *an interpretation. If* i *is the* fixpoint *of one iteration of forward chain, then* i *is a* model *for* p*.*

This is encoded as:

```
Lemma fwd_chain_sound def p i :
  safe_prog p → fwd_chain def p i = i → prog_true p i .
Proof.
move/allP⇒ h_sf /setUidPl/bigcups_seqP ⇒ h_cl ?.
by apply/allP⇒ ? h; apply: (cons_cl_sound (h_sf _ h)); apply: h_cl.
Qed.
```

Proof. By the same reasoning as above, the *fixpoint* condition becomes:

$$\forall \mathrm{cl},\ \mathrm{cl} \in \mathrm{p}, \texttt{cons\_clause}\ \mathrm{def}\ \mathrm{cl}\ \mathrm{i} \subseteq \mathrm{i}.$$

Since program safety implies the safety of each program clause, the conclusion follows directly from Theorem 8.5.8. □

Finally, based on Theorem 8.5.9 and Theorem 8.5.10, we establish that:

```
Lemma fwd_chainP def p i (p_safe : prog_safe p) :
  reflect (prog_true p i) (fwd_chain def p i == i).
```

8.6. Characterization of the Positive Engine

The main results presented in this section are the *termination* of the positive engine's evaluation (see Section 8.6.1) and its *soundness* and *completeness* (see Section 8.6.2).

8.6.1. Fixpoint Properties

In order to prove the bottom-up evaluation implemented by the positive engine terminates, we rely on properties of its forward chain operator. Specifically, we show that it is *monotonous*, *increasing* and *bounded*.

Monotonicity We prove forward chain is monotonous, by showing all its base functions (see Section 8.4.1) have the property. The first such function is the monadic bind `bindS`, implementing set flattening. The corresponding monotonicity lemma is:

```
Lemma bindS_mon {A B : finType} :
  ∀(i1 i2 : {set A}) (f1 f2 : A → {set B}),
  i1 ⊆ i2 → (∀x, f1 x ⊆ f2 x) →
  bindS i1 f1 ⊆ bindS i2 f2.
```

The proof relies on the `subsetP` and `bindP` reflection lemmas from the `fintype` library:

```
Lemma bindP {A B : finType} :
  ∀(i : {set A}) (f : A → {set B}) (r : B),
  reflect (exists2 s : A, s ∈ i & r ∈ f s) (r ∈ bindS i f).

Lemma subsetP {T : finType} (A B : pred T):
  reflect {subset A ≤ B} (A ⊆ B).
```

We prove monotonicity of the monadic set fold, as a direct consequence of `bindS_mon`.

```
Lemma foldS_mon {A : eqType} {B : finType}
  (f1 f2 : A → B → {set B}) (l : seq A)
  (f_mon : ∀x y, f1 x y ⊆ f2 x y) :
  (∀(s1 s2 : {set B}), (s1 ⊆ s2) → foldS f1 s1 l ⊆ foldS f2 s2 l).
```

Based on this, we establish monotonicity of the matching functions, building up to the logical consequence operator `cons_cl`.

```
Lemma match_atom_all_mon i1 i2 s a : i1 ⊆ i2 →
  match_atom_all i1 a s ⊆ match_atom_all i2 a s.

Lemma match_body_mon i1 i2 cl : i1 ⊆ i2 →
  match_body (body_cl cl) i1 ⊆ match_body (body_cl cl) i2.
```

```
Lemma cons_cl_mon i1 i2 cl def : i1 ⊆ i2 →
  cons_clause def cl i1 ⊆ cons_clause def cl i2.
```

These properties trivially follow by compositionality, from the `imsetS` and `preimsetS` properties in `finset`, stated below.

```
Variables (T1 T2 : finType) (f : T1 → T2).

Lemma imsetS (A B : T1) : A ⊆ B → (f A) ⊆ (f B).
Lemma preimsetS (A B : T2) : A ⊆ B → (f^-1 A) ⊆ (f^-1 B).
```

Using the `finset` library was particularly convenient, as it spared us from having to do routine (yet typically much larger) induction proofs on sets or on their list of elements up to permutation.

Finally, we establish monotonicity of the one-step forward chain:

```
Lemma fwd_chain_mon i1 i2 p def :
  i1 ⊆ i2 → fwd_chain def p i1 ⊆ fwd_chain def p i2.
```

The proof additionally uses the `bigcupP` characteristic property of the big union operator, from the `bigop` library:

```
Lemma bigcupP (T I : finType) (x : T) (P : pred I) :
  ∀ (F : I → {set T}), reflect (exists2 i : I, P i & x ∈ F i)
                               (x ∈ (\bigcup_(i | P i) F i)%SET).
```

Increase and Boundedness These properties are stated as follows:

```
Lemma fwd_chain_inc i p def : i ⊆ fwd_chain def p i.
Lemma fwd_chain_bound s : s ⊆ bp → fwd_chain def p s ⊆ bp.
```

The first proof trivially follows since the operator is inflationary by definition. Concerning the second result, textbook proofs, such as the one in [1], construct the bound by taking the program's Herbrand base, denoted as $\mathbf{B}(\mathbf{P})$. However, we did not need to explicitly do so, as it sufficed to take the `setT`, the top element of sets over a finite type, as the upper bound:

```
Definition bp : {set gatom} := setT.
```

As such, it was convenient that we could express the type of ground atoms as being finite. Also, note that the textbook construction on $\mathbf{B}(\mathbf{P})$ is only required when proving we can restrict to the active domain (see Section 8.1.1).

Having proved the one-step forward chain is *monotonous*, *increasing* and *bounded*, establishing its iteration reaches a (minimal) fixpoint (`lfpE` and `min_lfp_all`) is a corrolary of a more general Knaster-Tarski result (see Section 5.2.2). To this extent we were able to fully reuse the corresponding proof by contradiction, from [33].

```
Variables (T : finType) (s0 ub : {set T}).
Implicit Types (s : {set T}) (f : {set T} → {set T}).
Variables (f : {set T} → {set T}).

Definition ubound := ∀s, s ⊆ ub → f s ⊆ ub.

Hypothesis (f_mono : monotone f) (f_inc : increasing f)
           (f_ubound: ubound).

Notation iterf_incr n := (iterf f s0 n).
Notation lfp_incr     := (iterf f s0 #|ub|).

Hypothesis (s0_bound : s0 ⊆ ub).

(* boundedness of iteration is proved by induction *)
Lemma iterf_incr_bound n : (iterf_incr n) ⊆ ub.
Proof. by elim: n ⇒ /= [|n ihn]; rewrite ?lb_bound ?f_ubound //=.
Qed.

Lemma lfpE : lfp_incr = f lfp_incr.

Lemma min_lfp_all s (hs : s0 ⊆ s) (sfp : s = f s) : lfp_incr ⊆ s.
Proof. by rewrite (fix_iter #|ub| sfp); apply/iter_mon. Qed.

Lemma has_lfp :
  { lfp : {set T} | lfp = f lfp
                  ∧ lfp = iter #|ub| f s0
                  ∧ ∀s', s0 ⊆ s' → s' = f s' → lfp ⊆ s'}.
Proof.
by ∃lfp_incr; rewrite -lfpE /lfp_incr; repeat split;
apply/min_lfp_all.
Qed.
```

8.6.2. Strong Soundness and Completeness

Theorem 8.6.1. *Let* p *be a* safe *standard* Datalog *program. The iterative forward chain inference engine* terminates *and outputs a* minimal *model* m.

Proof. The fact that the engine iterating one-step forward chain reaches a minimal fixpoint follows from the `has_lfp` lemma in the previous section. It remains to show that m is indeed the minimal model for p. The fact that $m \models p$ is a direct consequence of the *forward chain soundness* lemma `fwd_chain_sound` (see Theorem 8.5.10). The minimality of the model follows from the minimality of the fixpoint, as the *forward chain stability* property (see Theorem 8.5.9) ensures $\forall m', m' \models p \Rightarrow$ `fwd_chain` def p $m' = m'$. □

The corresponding Coq statement is given below. Note that imposing program safety is required, as the condition is needed in the soundness proof of one step forward chain.

```
Section Completeness.

Variables (n : nat) (ct : finType) (def : constant ct).
Variables (st : finType) (ar : {ffun st → nat}).
Variable (p : program n ct ar).

Hypothesis p_safe : prog_safe p.
Notation gatom := (gatom ct ar).
Definition bp : {set gatom} := setT.

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
              & ∀ (m' : {set gatom}), s0 ⊆ m' →
                prog_true p m' → m ⊆ m']}}.

(* corollary to the previous technical lemma *)
Lemma incr_fwd_chain_complete_gen :
{ m : {set gatom} &
{ n : nat | [∧ prog_true p m
              , n = #|bp|
              , m = iter n (fwd_chain def p) set0
              & ∀ (m' : {set gatom}), prog_true p m' → m ⊆ m']}}.

End Completeness.
```

8.7. Discussion

As mentioned in Section 8.1.3, we established the key theorems proved this section also in an alternative formalization, in which we did not make the finiteness assumption for the domain of constants. The main consequence - apart from larger proof sizes for equivalent lemmas - was that, given a program p, we had to explicitely construct its bound, i.e its Herbrand base $\mathbf{B}(\mathbf{P})$, and prove it is indeed a model of p, i.e, $\mathbf{B}(\mathbf{P}) \models \mathtt{p}$. In the following, we will detail the most important differences that appeared in the *infinite*-state setting, concerning both modelization and proofs.

At the program signature level, we encode the domain of constants and the symbols of DATALOG programs as types with the choice operator (with decidable equality). With regard to the modeling of (ground) atoms, these take the form:

```
Variable symtype  : choiceType.
Variable constype : choiceType.
Variable arity    : symtype → nat.

Inductive constant := C of constype.

Inductive gatom := GAtom of symtype & seq constant.
Inductive  atom := Atom  of symtype & seq constant.

Definition wf_gatom ga :=
  arity (sym_gatom ga) ≡ size (arg_gatom ga).
Definition wf_atom  a  := arity (atom_sym a) ≡ size (atom_arg a).
```

Note that the well-formedness proofs for (ground) atoms are encoded separately in this setting, i.e, are not part of the type for ground atoms, as in Section 8.2.2. Also, the rest of the language constructs are formalized in the same manner as in the previous sections.

Building $\mathbf{B}(\mathbf{P})$ Let us assume a program p. We give a short overview of how we construct the Herbrand Base of p, i.e, the set of all possible ground atoms formed from its head symbols and active domain constants. First, we define the active domain, adom, as the list of all program constants prog_const.

```
Variable (p : program).

Definition term_const t : seq constant :=
  if t is Val e then [:: e] else [::].
```

```
Definition atom_const a : seq constant :=
  let: Atom _ args := a in
  flatten [seq term_const x | x ← args].

Definition tail_const tl : seq constant :=
  flatten [seq atom_const x | x ← tl].

Definition cl_const cl : seq constant :=
  tail_const [:: head cl & body cl].

Definition prog_const p : seq constant :=
  flatten [seq cl_const cl | cl ← p].

Definition adom : seq constant := prog_const.
```

To obtain a finite type corresponding to `adom`, we use the `seq_sub` function from the `finType` library. For a type `T : choiceType` and a list `l : seq T` over `T`, `seq_sub T` is a new type `SeqSub {ssval : T, ssvalP : val ∈ l}`, packing an element `ssval` of `T` and a (boolean) proof `ssvalP` that `ssval` belongs to `l`. This record type has the important property it possesses a canonical `finType` instance. Also, we can coerce `seq_sub T` to `T` using the (injective) projection `ssval`.

```
Definition strip_adom (t : seq (seq_sub adom)) : seq constant :=
  [seq ssval c | c ← t].
```

Next, we can construct the finite type containing tuples with arity `n` and with values in `adom`, i.e, `n.-tuple (seq_sub adom)`. Given an arbitrary type `T`, the type `n.-tuple T` of tuples over `T` with arity `n` is defined in SSReflect as a subtype of lists over `T`, i.e, `seq T`, that satisfies the size condition `size tval == n`. We can obtain the underlying tuple using the (injective) projection `tval`.

```
Structure tuple_of : Type :=
  Tuple {tval :> seq T; _ : size tval == n}.

Notation "{ 'tuple' n 'of' T }" := (n.-tuple T : predArgType)
  (at level 0, only parsing) : form_scope.
```

To enumerate the elements of the `n.-tuple (seq_sub adom)` type, we use the `enum` function from the `finType` library. Given a predicate `P` over a finite type, `enum P` builds the list with *all* elements of that type, satisfying `P`. Hence, `all_tuples n` gives us the desired list of tuples. Mapping over it the composition of the projection functions `tval` and `ssval`, i.e, `strip_adom \o tval`, we can extract the corresponding lists of constants.

```
(* {: T } notation for predT : pred T *)
Definition all_tuples n := enum {: n.-tuple (seq_sub adom) }.
Definition all_adom   n := map (strip_adom \o tval) (all_tuples n).
```

For a given symbol `s`, the `bp_s s` function builds all the ground atoms having `s` as a symbol and arguments with values in `adom`. Finally, $\mathbf{B}(\mathbf{P})$ can be encoded with the `bp` function that flattens the result of applying `bp_s` to all the head symbols of the program.

```
(* All the instances of s(c1...cn) over constants in adom *)
Definition bp_s s : seq gatom :=
  [seq GAtom s t | t ← all_adom (arity s)].
Definition bp : seq_gatom := flatten [seq bp_s s | s ← sym_hd].
```

Note that we can also define $\mathbf{B}(\mathbf{P})$ directly, using the `refine` tactic, as shown next.

```
Definition bp : seq gatom.
refine (flatten _).
refine ([seq _ | s ← sym_hd]).
set adomT := seq_sub adom.
set argT := (arity s).-tuple adomT.
set l := enum {: argT}.
set l' := [seq val t | t ← l].
rewrite /= in l'.
exact : [seq GAtom s [seq val c | c ← t] |t ← l'].
Defined.
```

Finally, we establish the characteristic property `bpP` for `bp`. This states that, a ground atom `ga` is well-formed, has a symbol belonging to the head symbols of the program and arguments with constants in `adom` if and only if it belongs to `bp`.

```
Lemma bpP ga :
  [∧ wf_gatom ga, sym_gatom ga ∈ sym_hd
    & {subset arg_gatom ga ≤ adom} ] ↔ ga ∈ bp.
```

Proving $\mathbf{B}(\mathbf{P})$ is a program model We now prove $\mathbf{B}(\mathbf{P})$ is a model of any safe program, whose clause heads are well-formed. That is to say, `bp` $\models$ `p`, as stated in the `bpM` lemma. Note that the `seq_fset` function transforms a given list into a set containing its elements.

```
Definition prog_heads p    := [seq (head cl) | cl ← p].
```

```
Definition wf_prog_heads p := all wf_atom (prog_heads p).


Variable (p_wf : wf_prog_heads p) (p_safe : prog_safe p).

Lemma bpM : prog_true p (seq_fset bp).
```

The Coq proof follows the paper version given in [1]. Let $\bar{H} \leftarrow \bar{B}_1, \ldots, \bar{B}_n$ be the instantiation of a program clause $H \leftarrow B_1, \ldots, B_n$. The proof reduces to showing $\mathbf{B}(\mathbf{P}) \models \bar{H} \leftarrow \bar{B}_1, \ldots, \bar{B}_n$, i.e, to showing that: if $\{\bar{B}_1, \ldots, \bar{B}_n\} \subseteq \mathbf{B}(\mathbf{P})$, then $\bar{H} \in \mathbf{B}(\mathbf{P})$. According to `bpP` this implies showing that: 1) $\bar{H}$ is well-formed, 2) the symbol of $\bar{H}$ is among the program symbols and 3) the arguments of $\bar{H}$ are in `adom`. The first two subgoals follow trivially from the hypothesis. We are left with proving the last one. From the safety condition, we know that the variables in H are among those in the body, i.e, $Var(H) \subseteq Var(B_1) \cup \ldots \cup Var(B_n)$. Since, $\{\bar{B}_1, \ldots, \bar{B}_n\} \subseteq \mathbf{B}(\mathbf{P})$, all body constants belong to `adom`. By an argument analogous to the one given in the proof of Theorem 8.1.1, it follows that each constant occuring in $\bar{H}$ is in `adom`. Hence, $\bar{H} \in \mathbf{B}(\mathbf{P})$.

Proving $\mathbf{B}(\mathbf{P})$ is a program bound Since $\mathbf{B}(\mathbf{P})$ is a program model, the fact that it is a bound for bottom-up inference follows from the stability property of one-step forward chain (see Theorem 8.5.9).

Proving Strong Soundness and Completeness of Forward Chain Finally, we establish the analogous result to that proven in Theorem 8.6.1. That is to say:
"Let p be a *safe* standard (positive) Datalog program. The iterative forward chain inference engine *terminates* and outputs a *minimal* model m."

The proof is similiar to the one given is Section 8.6.2, except for the explicit bound constructed for the positive engine, i.e, for the iteration of the forward chain operator. Note that `fset0` is a notation for the empty set in the `finmap` library.

```
Lemma fwd_chain_complete_alt :
{ m : {fset gatom} &
{ n : nat | [∧ prog_true p m
              , m = iter n (fwd_chain def p) fset0
              & ∀m', prog_true p m' → m `≤` m']}}.
Proof.
have h_mon := fwd_chain_mon p def.
have h_fp  := fwd_chain_stable def (bpM p_wf p_safe).
have h_ub  : bounded bp (fwd_chain def p).
  by move⇒ ss H; rewrite -h_fp; apply/h_mon.
have [m [m_fix [m_def m_min]]] := has_lfp h_mon h_ub.
∃m, #|`bp|; do ! split; auto; first exact/(fwd_chain_sound p_safe (
    esym m_fix)).
```

```
by move⇒ m' /fwd_chain_stable /esym /m_min.
Qed.
```

8.7.1. Contributions

In this chapter we presented a Coq/ssreflect formalization of the theory of positive Datalog, based on the Mathematical Components library for sets over finite types, i.e, `finset`. The formalization consists of the following components:

- the positive Datalog language and its model-theoretic semantics
- a corresponding inference engine implementing bottom-up evaluation:
  - development of a core theory of substitutions
  - development of a monadic abstract matching algorithm
- the fixpoint semantics of the language
  - set-theoretic definitions of the immediate consequence and fixpoint operators
  - monotonicity, boundedness and fixpoint results
- proofs of the characteristic properties of the engine:
  - subsumption, injectivity, decomposition, modularity and stability
  - main theorems: soundness, termination and completeness of the evaluation with respect to the model-theoretic semantics

8.7.2. Lessons

The matching algorithm employed by the engine iteratively tries to construct groundings from substitutions. As such, an important aspect in reasoning about its soundness and completeness consists in reasoning about substitutions, i.e, about their domains, orderings, properties and application to different language constructs. To this end we formalized a small theory for substitutions, mirroring standard logic approaches (see [21]) to characterizing hyperresolution.

An interesting point is that the completeness proof for matching body atoms against interpretations (ground atom sets) was needed for proving the soundness of the forward chain procedure. Regarding the completeness proof for the latter, we found it easier to follow a fixpoint theory approach, rather than to rely on lifting and compactness arguments. In particular, we used a Knaster-Tarski result to prove that the evaluation output was indeed a minimal fixpoint (due to the monotonicity and boundedness of the fixpoint operator). Also, we needed to establish the "stability" of the fixpoint operator, i.e, the fact that, for a given program, its models are fixpoints.

9. Formalization of Datalog with Negation

In this chapter we discuss a formalization of DATALOG with negation, based on the one we developed in Section 8, for standard DATALOG. The library contains a mechanization of the language and of its semantics, together with that of an inference engine, implementing the stratified evaluation heuristic. It is complete with proofs capturing the characteristic properties of all defined components. We briefly summarize below the main idea behind the functioning of the engine, which lies at the core of the development.

As we have seen in Section 6, logic programs with negation are more difficult to handle than non-negated programs, as, in particular, it is not guaranteeed that an (unique) minimal model always exists. To remedy this, a sufficient condition for model existence is provided in the literature, namely *stratification*. The main idea of the technique is to segment a logic program in such a way that every slice can be soundly handled as a positive one and, as such, its minimal model can be computed and iteratively enriched as we move to higher stratas. Not all programs are stratifiable, but, for those that are, a stratification can be defined by computing predicate dependencies. Thus, the lowest strata must forcefully be a negation free, self-contained slice of the original program. Due to the self-containement property, later layers can rely on the fixpoint semantics given in Section 5. The COQ development is impacted by stratification in two ways: first, the computed models have to be indexed by stratas and second, every slice of the negated program has to be seen as a positive program, which is realized via an embedding described in Section 9.2.1.

The chapter is organized as follows. We start with Section 9.2, by outlining the relevant modelization choices we made. Next, in Section 9.1.1 and Section 9.1.2, we present the syntax and semantics of the language. In Section 9.3, we give a formalization of relevant aspects from the theory of stratification. In Section 9.4, we outline our mechanization of stratified evaluation. In Section 9.5, we review the proofs of additional characteristic properties we had to provide, in order to reuse the inference engine from Section 8. Also, we introduce the essential properties that the evaluation of programs with negation has to satisfy at each step, i.e, its invariant. This is a prelude to Section 9.6, in which we conclude with the main proof, establishing stratified evaluation to be *sound*, *complete* and *terminating* with a *"relatively" minimal model*.

9.1. Language Representation

We give an account of a formalization of the syntax (see Section 9.1.1) and semantics (see Section 9.1.2) of DATALOG programs with negation.

9.1.1. Syntax

We extend the syntax of positive DATALOG with *literals* that, according to Section 6, are positive or negative atoms. Consequently, we reuse the definitions of ground (`gatom`) and non-ground atoms (`atom`), from Section 8.2. Also, as before, we distinguish between ground and non-ground literals and clauses.

Literals Literals are encoded by enriching ground and, respectively, non-ground atoms with a boolean flag, marking whether or not they are negated.

```
Inductive glit := GLit of bool * gatom.
Inductive lit  := Lit  of bool * atom.
```

Clauses Ground clauses are defined as packing ground atoms and ground literal lists, while non-ground ones pack atoms and literal lists. Note that, in both cases, we did not need to model (ground) clause heads with (ground) literals. This is due to the fact that, as mentioned in Section 6, within a clause, negation only occurs in the body.

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

The formalization of programs as clause lists is the same as in Section 8.2.

Safety The *safety condition* imposed on DATALOG programs with negation is the same as that described in Section 9.1.1. The *extended safety condition* from Definition 6.1.4 - stating that, within each program clause, variables in negated body atoms, should also appear in positive ones - is not needed. This is due to the fact that the evaluation algorithm we define in Section 9.4.3 operates on *stratifiable* programs. As noted in Remark 6.2.10, these have the property that they can be partitioned into semipositive programs, i.e, programs whose negated atoms can be complemented with respect to their active domain. Since the latter is finite, we do not need any further restriction.

```
Definition term_vars := P.term_vars.
Definition atom_vars := P.atom_vars.

Definition lit_vars  l  := atom_vars (atom_lit l).
Definition tail_vars tl := \bigcup_(t ← tl) lit_vars t.

Definition cl_safe  cl :=
  atom_vars (head cl) ⊆ tail_vars (body cl).

Definition prog_safe p := all cl_safe p.
```

9.1.2. Semantics

The only additions to the interpretation of programs defined in Section 8.3.4 concern ground literals and clauses.

The `glit_true` definition below captures the fact that a ground literal `gl` is satisfied by an interpretation `i`, by casing on the ground literal's flag. If the latter is true, i.e the literal is positive, we check membership of the underlying ground atom in `i`; otherwise, i.e, if the literal is negative, validity in `i` is given by the absence of the underlying ground atom from `i`.

```
Definition glit_true i gl :=
  if flag_glit gl then gatom_glit gl ∈ i else gatom_glit gl ∉ i.
```

Satisfiability of a ground clause `gcl` with respect to an interpretation `i` holds if, when all body literals are satisfied by `i`, the head atom is as well.

```
Definition gcl_true gcl i :=
  all (glit_true i) (body_gcl gcl) ==> (head_gcl gcl ∈ i).
```

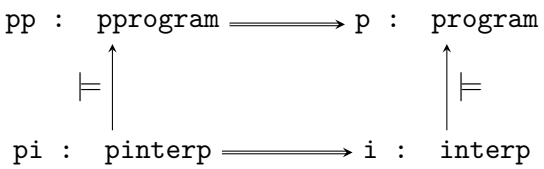
While using a different version of ground clause satisfiability, the definitions of non-ground clause and program satisfiability have the same form as in the positive case.

9.2. Positive Embedding

We present the modeling choices for the syntax (see Section 9.2.1) and semantics (see Section 9.2.2) aspects of employing the previous positive engine via positive embedding.

9.2.1. Syntax Aspects

To reuse the matching algorithm in Section 8.4.1, operating only on positive atoms, we define a "positive" embedding. This transforms the positive programs from Section 8.2, whose type we denote by `pprogram`, into programs supporting negation, as marked by an additional boolean flag, with which we decorate all atoms. This is done by *encoding* and, inversely, *uncoding* open and ground atoms, literals, clauses and programs *to* and *from* their positive counterparts. This syntactic embedding naturally extends to the semantic level, resulting in what we later refer to as *positive interpretations* (`pinterp`). In Section 9.4.2 we prove satisfiability is an invariant of program encoding, generalizing the commutativity of the diagram below :
Indeed, as the embedding is primarily used in the stratified evaluation part, we state the result in the context of *complemented interpretations* (`cinterp`). As detailed in Section 9.4.1, these pack normal interpretations (`interp`), together with their intended complement, with respect to the Herbrand base of programs slicings, from each iteration step. Consequently, satisfiability preservation is stated, depending on *well-complementation*



properties. We briefly review the way in which we set up the embedding and some of its properties, i.e, injectivity and safety invariance.

Symbol Embedding The type of predicate symbols `symtype` and their `arity` become `psymtype` and `parity`. Encoded symbols have a `true` flag, while uncoded ones have none.

```
Definition psymtype := [finType of bool * symtype].
Definition parity   := [ffun ps ⇒ arity ps.2].
Definition encodes s  := (true, s).
Definition uncodes ps := let: (_, s) := ps in s
```

A required (trivial) result is arity invariance with respect to encoding and uncoding. The proof directly follows from the `ffunE` pointwise function equality, a convenient property of finitely-supported functions (see Section 8.1.4).

```
Lemma arP  x : arity x  = parity (encodes x).
Lemma parP x : parity x = arity  (uncodes x).
```

Atom Embedding The embedding of an `atom` relies on that of its underlying type `raw_atom` and is the most technical of all construct embeddings leading up to programs. Having imported the corresponding types for positive (raw) atoms, i.e, `gen_ratom` and `gen_atom`, we define a "generic" raw atom embedding `map_raw_atom_sym`, obtained by applying a function `f` to its symbol. In `map_raw_atom_proof`, we prove that, when `f` is arity-preserving, the raw atom embedding of a positive atom (well-formed by construction) is well-formed. Consequently, we define the "generic" atom embedding `map_atom_sym`:

```
(* we import the module containing the library developed for
   positive Datalog *)
Module P := pengine.
Variable constype : finType.

Notation gen_ratom := (P.raw_atom n constype).
```

```
Notation gen_atom  := (P.atom n constype).


Definition map_raw_atom_sym
  (st1 st2 : finType) (f : st1 → st2) (gena : gen_ratom st1) :
  gen_ratom st2 := P.RawAtom (f (sym_atom gena)) (arg_atom gna).

Lemma map_raw_atom_proof (st1 st2 : finType)
  (ar1 : {ffun st1 → nat}) (ar2 : {ffun st2 → nat})
  (f : st1 → st2) (h : ∀x, ar1 x = ar2 (f x)) (gna : gen_atom ar1)
  : P.wf_atom ar2 (map_raw_atom_sym f gna).

Definition map_atom_sym (st1 st2 : finType)
  (ar1 : {ffun st1 → nat}) (ar2 : {ffun st2 → nat})
  (f : st1 → st2) (h : ∀x, ar1 x = ar2 (f x)) (gna : gen_atom ar1)
  := P.Atom (map_raw_atom_proof h gna).
```

This is then instantiated with the corresponding-preservation proofs, i.e, `arP` and `parP`, to express the encoding and uncoding of atoms, i.e, `encodea` and `uncodea`.

```
Definition encodea  := map_atom_sym  arP.
Definition uncodea  := map_atom_sym parP.
```

Literal Embedding The embedding of literals trivially extends that of atoms. If a given literal has an explicitly negative flag, i.e, represents a negated atom, it can be encoded into a positive one, by flipping the (true by default) boolean flag in the encoding of its underlying atom. The uncoding of positive atoms is done by making their boolean flag explicit in constructing the corresponding literals:

```
Definition flips ps := let: (b, s) := ps in (∼ ∼ b, s).
Lemma farP x : parity x = parity (flips x).
Definition flipa  := map_atom_sym farP.

Definition encodel (l : lit) : patom :=
  let: Lit (b, a) := l          in
  let: pa         := encodea a in
  if b then pa else flipa pa.

Definition uncodel (pa : patom) : lit :=
  let: a := uncodea pa in
  if flag_psym (sym_atom pa) then Lit (true, a) else Lit (false, a).
```

Note that $\sim\sim$ is a construct from the SSReflect library `ssrbool` and that it corresponds to boolean negation.
The embedding of ground atoms and literals is analogous to the above definitions. Also, we can easily extend these to capture the encoding and uncoding of programs. In the corresponding Coq proofs, we denote the encoding of a program `p` as ⌜p⌝.

Injectivity Based on the proofs that literal encoding and uncoding are inverse with respect to each other and, hence, that they are injective, we prove the analogous cancelation lemmas for clauses. The only difference is that, in this setting, the encoding of clauses is inverse to their uncoding and, hence is injective, only when the flag of the head clause's encoded atom is positive, i.e, when `hd_cl_pos` is true. The injectivity property is required for establishing that, if a program with negation is satisfied by a given complemented interpretation that is well-complemented, then its encoding is satisfied by the corresponding positive interpretation.

```
Definition hd_cl_pos pcl := flag_psym (P.hsym_cl pcl).

Lemma encodeclK : cancel encodecl uncodecl.
Lemma uncodeclK : {in hd_cl_pos, cancel uncodecl encodecl}.
```

Safety Invariance Additionally, it is trivial to prove that program safety is invariant with respect to program encoding.

```
Lemma encode_prog_safe p : prog_safe p = P.prog_safe (encodep p).
Proof. by rewrite/P.prog_safe all_map; apply/eq_all/encode_cl_safe.
Qed.
```

9.2.2. Semantics Aspects

The main modeling choice at the semantics level consists of introducing complemented interpretations, to *explicitly* account for the negative facts that are true, at every evaluation step. A complemented interpretation (see Section 9.4.1) extends a given interpretation with its complement, with respect to the set of all possible ground atoms that can be constructed from its symbols. We illustrate them below, in the context of reusing the positive engine's evaluation.

Example 9.2.1. *Let us revisit Example 6.2.15, in which the program P is stratified by the strata consisting of $\{r, s, t\}$, $\{q\}$ and $\{p\}$ into the program slices P_1, P_2 and P_3. Their corresponding positive encodings are $\ulcorner P_1 \urcorner$, $\ulcorner P_2 \urcorner$ and $\ulcorner P_3 \urcorner$:*

$$P_1 = \left\{ \begin{array}{l} r(X) \leftarrow t(X) \\ s(b) \leftarrow \\ t(a) \leftarrow \end{array} \right\} \qquad \ulcorner P_1 \urcorner = \left\{ \begin{array}{l} (\top, r)(X) \leftarrow (\top, t)(X) \\ (\top, s)(b) \leftarrow \\ (\top, t)(a) \leftarrow \end{array} \right\}$$

$$P_2 = \left\{ \begin{array}{l} q(X) \leftarrow s(X), \neg t(X) \\ q(a) \leftarrow \end{array} \right\} \qquad \ulcorner P_2 \urcorner = \left\{ \begin{array}{l} (\top, q)(X) \leftarrow (\top, s)(X), (\bot, t)(X) \\ (\top, q)(a) \leftarrow \end{array} \right\}$$

$$P_3 = \left\{ \begin{array}{l} p(X) \leftarrow \neg q(X), r(X) \\ p(X) \leftarrow \neg t(X), q(X) \end{array} \right\} \qquad \ulcorner P_3 \urcorner = \left\{ \begin{array}{l} (\top, p)(X) \leftarrow (\bot, q)(X), (\top, r)(X) \\ (\top, p)(X) \leftarrow (\bot, t)(X), (\top, q)(X) \end{array} \right\}$$

We begin the stratified evaluation of P by computing the model M_1 for $\ulcorner P_1 \urcorner$, using the positive inference engine: $M_1 = T_{\ulcorner P_1 \urcorner} \uparrow \omega(\emptyset) = \{(\top, r)(a), (\top, s)(b), (\top, t)(a)\}$. Next, complementing the interpretation with respect to the Herband base $\mathbf{B}(\ulcorner \mathbf{P_1} \urcorner)$, we obtain the complemented interpretation:

$$(\{(\top, r)(a), (\top, s)(b), (\top, t)(a)\}, \{(\bot, r)(b), (\bot, s)(a), (\bot, t)(b)\}).$$

This will be passed to the positive engine, as a single positive interpretation, in the second evaluation phase. Consequently,

$$M_2 = T_{P_2} \uparrow \omega(\{(\top, r)(a), (\top, s)(b), (\top, t)(a), (\bot, r)(b), (\bot, s)(a), (\bot, t)(b)\}).$$

M_2 adds to M_1 two new facts, i.e, $\{(\top, q)(a), (\top, q)(b)\}$. Hence, the new complemented interpretation is:

$$(\{(\top, r)(a), (\top, s)(b), (\top, t)(a), (\top, q)(a), (\top, q)(b)\}, \{(\bot, r)(b), (\bot, s)(a), (\bot, t)(b)\}).$$

Passing it to the positive engine, in the final evaluation phase, results in:

$$M_3 = T_{P_3} \uparrow \omega(\{(\top, r)(a), (\top, s)(b), (\top, t)(a), (\top, q)(a), (\top, q)(b), (\bot, r)(b), (\bot, s)(a), (t, \bot)(b)\}).$$

M_3 adds to M_2 one new fact, i.e, $\{(p, \top)(b)\}$. The current complemented interpretation (not well-complemented *though) is:*

$$(\{(\top, r)(a), (\top, s)(b), (\top, t)(a), (\top, q)(a), (\top, q)(b), (\top, p)(b)\}, \{(\bot, r)(b), (\bot, s)(a), (\bot, t)(b)\}).$$

The set of all the derived positive facts is obtained "sanitizing" the above, by projecting its first component:

$$\{(\top, r)(a), (\top, s)(b), (\top, t)(a), (\top, q)(a), (\top, q)(b), (\top, p)(b)\}.$$

Hence, the final model is $M(P) = \{r(a), s(b), t(a), q(a), q(b), p(b)\}$.

9.3. Stratification

We model *stratifications* based on the notion of a "strata", i.e, a list of symbol sets, partitioning the set of symbols of a given DATALOG program with negation. Implicitly, we assume the first list element to be the lowest stratum and the last to be the upper one.

```
Definition strata := seq {set symtype}.
```

We present a mechanized strata characterization in Section 9.3.1 and, in Section 9.3.3, a characterization of the induced program slicings.

9.3.1. Strata Characterization

There are many possibilities for capturing required strata properties, as seen in Section 6. The following emerged as sufficient conditions for completing the proof of our main theorem, described in Section 9.6.1:

1. **each strata should refer to a *disjoint* set of symbols**
   (disjointness condition)
2. **stratum symbols *cannot be refered to* negatively[1] in lower or equal strata**;
   equivalently, they *can only refer to* negated symbols defined in strictly lower strata
   (negative-dependency condition)
3. **stratum symbols *only* depend on[2] symbols from lower or equal strata**;
   equivalently, they are *independent* from strictly upper strata, and, consequently, program slices can be evaluated separately, from the bottom-up
   (positive-dependencies condition/self-containment)

Given that we represent strata as lists, it seems natural to use a recursively defined predicate for capturing the above invariants. This matches the structure of the main algorithm, which iterates on the strata list. The technical definition is given in the `is_strata_rec` fixpoint predicate below. The key point is the use of an accumulator, such that we can refer to already processed stratas; this is typical of functional programming. We also require that all program symbols be present in the strata, as enforced by the empty list base case. Next, we discuss the step case conditions.

Let `p` be a normal program. At an intermediate processing stage, a full stratification of `p` consists of a *current strata* `str` that has at least one stratum, namely `ss`, and an *accumulated strata* `acc`, containing the list of previously visited sets of symbols.

[1] We say a stratum symbol is refered to negatively in a program, if it corresponds to a negated atom in the body of a clause from the program.

[2] We say that a symbol s_1 depends on a symbol s_2, if s_1 appears in the body of a clause having s_2 as head symbol.

1. The *disjointness condition* above is implied by the *local* disjointness condition expressed via the `[disjoint acc & ss]` predicate. This states that `ss` symbols do not appear in any other `acc` strata.

2. The *negative-dependency condition* is encoded using the `negdep` predicate, which holds for a symbol `s`, given a program `p`, if `s` is negated in the body of a `p` clause:

```
(* flattening of clause body literal lists for clauses in p *)
Definition lits_prog p := flatten [seq body_cl cl | cl ← p].

(* list of all program literals with a negative flag *)
Definition nlit_prog p :=
  [seq l ← lits_prog p | ~~ flag_lit l].

(* membership in the negative literal symbol list *)
Definition negdep  s p :=
  s ∈ [seq sym_lit l | l ← nlit_prog p].
```

   As such, all the symbols in the current stratum `ss` are forbidden from being refered to negatively (appearing negated in clause bodies) in the program sliced up to `ss`:

```
all (predC (negdep^~(slice_prog p (acc ∪ ss)))) (enum ss),
```

   which is a notation for:

```
all (predC (fun x ⇒ negdep x (slice_prog p (acc ∪ ss)))) (enum ss)
```

   and, respectively, for:

```
all (fun x ⇒ ~~ negdep x (slice_prog p (acc ∪ ss))) (enum ss),
```

3. The *positive-dependency/self-containment condition* is encoded using the `posdep` predicate that holds for a stratum `ss`, given a clause `cl`, if all body symbols in `cl` appear in `ss`.

```
(* list of symbols for literals from a clause body *)
Definition bsym_cl cl   := [seq sym_lit l | l ← body_cl cl].
Definition posdep ss cl := bsym_cl cl ⊆ ss.
```

   The condition states that all clause body symbols from the slicing of the program with strata up to the current one appear in the latter.

```
all (posdep (acc ∪ ss)) (slice_prog p (acc ∪ ss)).
```

Initializing the auxiliary strata predicate `is_strata_rec` with the empty set `set0`, we can finally define the `is_strata` predicate, for a given program `p` and a strata `str`.

```
Fixpoint is_strata_rec p str acc :=
  match str with
  | [::]      ⇒ [set x in sym_prog p] ≡ acc
  | ss :: str ⇒ [&& is_strata_rec p str (acc ∪ ss)
                  , [disjoint acc & ss]
                  , all (predC (negdep^~(slice_prog p (acc ∪ ss))))
                        (enum ss)
                  & all (posdep (acc ∪ ss))
                        (slice_prog p (acc ∪ ss))
                ]
  end.

Definition is_strata p str := is_strata_rec p str set0.
```

9.3.2. Exhaustive Stratification Computation

Due to symbol type finiteness, we can give an effective, albeit inefficient, algorithm for computing a stratification satisfying properties from the previous section. We define the type of stratifications (`strat_type`) as packing an ordinal, bounded by the maximal number of program symbols, and a tuple of that length, consisting of sets of symbols, i.e the strata.

```
Definition strata_type :=
  { s_l : 'I_#|symtype|.+1 & s_l.-tuple {set symtype} }.
```

To ensure we indeed have a finite number of strata, we impose the additional condition `is_strata_strong`. This requires that all symbols sets of a strata be non-empty.

```
Definition is_strata_strong p str :=
  is_strata_rec p str set0 && all (predC1 set0) str.
```

We model the requirement using the `predC1` operator, from the `SSReflect` library. As such, the predicate `predC1 set0` denotes `[pred x | x != set0]`.

The `compute_strata` algorithm is defined as an option map `omap`. Given a program `p`, this extracts a potential corresponding stratification (the second projection `projT2` of the strata type), which obeys the `is_strata_strong` condition. This "witness" is computed using the SSReflect `[pick e | P e]` construct that picks an element `e`, satisfying a predicate `P`, among all inhabitants of the *finite* type of `e`. The soundness of the algorithm, i.e, `compute_strataP`, states that, if, for a program `p`, `compute_strata`

outputs a list of symbol sets `str`, then this is indeed a stratification for `p` and consists of non-empty strata. However, if the algorithm fails, then `p` is not stratifiable. The result follows from the characteristic property `pickP` of the `pick` operator and from the `negb_exists` lemma. The latter expresses that, for elements `x` of finite type, `~~ [∃x , P x] = [∀x, ~~ P x]`. It is the encoded, boolean version of the logic proposition $\neg(\exists x, Px) = \forall x, \neg Px$, which exceptionally[3] holds precisely because the type of `x` is finite.

```
Definition sttl (x : strata_type) : seq {set symtype} := projT2 x.
Definition compute_strata p : option strata :=
  omap sttl [pick x : strata_type | is_strata_strong p (sttl x)].

Lemma compute_strataP p :
  if compute_strata p is Some str
  then is_strata_strong p str
  else ~~ [∃x : strata_type, is_strata_strong p (sttl x)].
Proof.
rewrite /compute_strata; case: pickP ⇒ //= hs.
by rewrite ¬_exists; apply/forallP⇒ s; rewrite hs.
Qed.
```

9.3.3. Slicing Characterization

Given a program p and an interpretation i, we consider a stratification, as previously defined, and one of its arbitrary stratum elements ss. The latter induces, at a syntactic level, a *program slicing* and, at the semantic level, an *interpretation slicing*. Both are implemented using *filter functions*, as follows.

Program Slicing We call the *slicing* of p with ss, denoted as p_{ss}, the list of all clauses whose head symbols appear in ss. Equivalently, relating this to Definition 6.2.7, $p_{ss} = \bigcup_{s \in ss} def(s)$. This is encoded as:

```
Definition slice_prog p ss := [seq cl ← p | hsym_cl cl ∈ ss].
```

The corresponding *characterization* is given by `slice_progP`. This states that, for a program `p` and a stratum `ss`, the set of head symbols of the program obtained by slicing `p` with `ss`, i.e, `hsym_prog (slice_prog p ss)` is a subset of the latter. We model the lemma using the extensional, propositional characterization of the subset relation, i.e, for sets `A` and `B` over a type `T`, `{subset A ≤ B}` denotes `∀x : T, x ∈ A → x ∈ B`.

[3] it is equivalent to the excluded-middle property, not guaranteeed by default in intutionistic logic

```
(* list of head symbols of a clauses and programs *)
Definition head_cl cl := let: Clause h b := cl in h.
Definition hsym_prog p := [seq hsym_cl cl | cl ← p].

Lemma slice_progP p ss : {subset hsym_prog (slice_prog p ss) ≤ ss}.
Proof. by move⇒? /mapP[?]; rewrite mem_filter; case/andP⇒?? →.
Qed.
```

The `slice_progU` lemma gives the expected *decomposition* property, namely that the slicing of a program `p` with the union of two symbols sets `ss1` and `ss2` is *extensionally equal* to the concatenation of the slicing of `p` with `ss1` and of its slicing with `ss2`.

```
Lemma slice_progU p ss1 ss2 :
  slice_prog p (ss1 ∪ ss2) =i slice_prog p ss1 ++ slice_prog p ss2.
Proof. by move⇒ ?; rewrite mem_cat !mem_filter !inE andb_orl. Qed.
```

Finally, the `sliced_prog_safe` lemma ensures that slicing preserves program safety.

```
Lemma sliced_prog_safe p ss :
  prog_safe p → prog_safe (slice_prog p ss).
Proof.
rewrite/slice_prog⇒ /allP h; apply/allP⇒ ?; rewrite mem_filter;
by case/andP⇒ _ ?; apply: h.
Qed.
```

Interpretation Slicing We call the *slicing* of i with ss, denoted as i_{ss}, the subset of i containing ground atoms, whose symbols appear in ss. This is encoded as:

```
Definition i_ssym ss i : interp := [set x in i | sym_gatom x ∈ ss].
```

The corresponding *characterization* is given by the `i_ssymP` reflection lemma.

```
Lemma i_ssymP ss i ga :
  reflect (ga ∈ i ∧ sym_gatom ga ∈ ss) (ga ∈ i_ssym ss i).
Proof. by apply/(iffP idP); rewrite !inE; case/andP; try split. Qed.
```

In the development, we also had to establish a series of results relating interpretation slicing and intepretation symbols to set theoretical operators. The proofs are based on properties of the finite set library and of the filter function. We exemplify with the most basic ones below.

```
(* composition property *)
Lemma i_ssymComp ss i : i_ssym ss (i_ssym ss i) = i_ssym ss i.
(* decomposition properties *)
Lemma i_ssymU ss i1 i2 :
  i_ssym ss (i1 ∪ i2) = i_ssym ss i1 ∪ i_ssym ss i2.
Lemma i_ssymI ss i1 i2 :
  i_ssym ss (i1 ∩ i2) = i_ssym ss i1 ∩ i_ssym ss i2.
Lemma i_ssymD ss i1 i2 :
  i_ssym ss (i1 \ i2) = i_ssym ss i1 \ i_ssym ss i2.
```

9.4. Stratified Evaluation

In this section we give the stratified evaluation algorithm for DATALOG with negation (see Section 9.4.3), presenting each of its building blocks. To this end, we start by introducing the notion of *complemented interpretation* (see Section 9.4.1), necessary due to the reuse of the inference engine from Section 8. Then, we establish the preservation of program satisfiability with respect to interpretation complementation and encoding (see Section 9.4.2).

9.4.1. Complemented Interpretations

To account for the negated facts that hold by absence from the model, we resort to a technical artifact and define a special type `cinterp` of *complemented interpretations*. This is a pair-type packing an interpretation and its complement with respect to `setT`, the top element of the `interp` finite type.

```
Notation cinterp := (interp * interp)%type.
```

Complementation As explained in Section 9.1.2, in evaluating a stratified program, we *incrementally* compute complemented interpretations for all strata sub-programs. Let us consider an intermediate iteration step, given a current stratum ss and a complemented interpretation ci, for previous sub-programs. To obtain a *new* complemented interpretation, also for the current sub-program, we have to complement ci with respect to ss, denoted as : C_{ss} ci $\equiv$ (ci.1, ci.1$^{\mathrm{C}_{\mathrm{ss}}}$ $\cup$ ci.2).

This is done in the `ciC` function below, extending `ci.2` with `ci.1`$^{\mathrm{C}_{\mathrm{ss}}}$, the finite set complement of `ci.1` sliced with `ss`, as computed by `ic_ssym` and `i_ssym`.

In our encoding, we used the complementation operator from the `finset` library. As presented below, for given a finite type `T`, and a set `A` over `T`, `~: A` is defined as the

complement of A. Since T is a finite type, we have a maximal set setT : {set T} with all the elements in T. Hence, ∼: A contains all elements of type T in setT, not in A.

```
Variable T : finType.
Definition setC (A : {set T}) := [set x | x ∉ A].
Notation "∼: A" :=
  (setC A) (at level 35, right associativity) : set_scope.

(* slicing of the complement of i with the set of symbols ss *)
Definition ic_ssym ss i := i_ssym ss (∼: i).
Definition ciC ss ci : cinterp := (ci.1, ic_ssym ss ci.1 ∪ ci.2).
```

Note that, due to the *inherent* properties of finite types, complemented interpretations have a *complete complemented lattice* structure. In this respect, the choice for their particular type encoding is an *essential* one, as proofs regarding complementation *naturally match* those corresponding to the ssreflect finite set complementation.

Well-Complementation A complemented interpretation ci is *well-complemented with respect to a set of symbols* ss - denoted wc_{ss} ci - if the elements of $\top^{\text{ss}}$ - the corresponding complemented lattice's top, consisting of *all* well-formed ground atoms with symbols in ss - are in *exactly one* ci component. Equivalently, the ci components *partition* $\top^{\text{ss}}$:

$$\text{ci.1}^{\text{ss}} \cup \text{ci.2}^{\text{ss}} = \top^{\text{ss}} \text{ and } \text{ci.1}^{\text{ss}} \cap \text{ci.2}^{\text{ss}} = \emptyset$$

This is encoded as:

```
Definition partf f A B S := (f A ∪ f B ≡ f S) && (f A ∩ f B ≡ ∅).
Definition ci_wc ss ci   := partf (i_ssym ss) ci.1 ci.2 setT.
```

A useful well-complementation property regards its decomposition with respect to union:

```
(* set of symbols of an interpretation *)
Definition isyms    i : {set symtype} := [set sym_gatom (val ga) |
   ga in i].
(* set of symbols of a complemented interpretation *)
Definition ci_syms ci : {set symtype} := isyms ci.1 ∪ isyms ci.2.

Lemma wcUP ss1 ss2 ci1 ci2 :
  ss1 ∩ ss2 = ∅ → ci_syms ci1 ⊆ ss1 → ci_syms ci2 ⊆ ss2 →
  ci_wc ss1 ci1 → ci_wc ss2 ci2 → ci_wc (ss1 ∪ ss2) (ci1 ∪ ci2).
```

From Complemented to Positive Interpretations – and Back As explained in Section 9.6.1, to evaluate a stratified program we reuse the positive engine mechanism from

Section 8, for each of the (encoded) program slices. Since running the positive engine on each encoding outputs a *positive interpretation* [4], while the main engine operates with *complemented interpretations*, it becomes necessary to relate the two.

Given a positive interpretation `pi : pinterp` and a complemented interpretation `ci : cinterp`, the corresponding transformations are given by the `c2p` and `p2c` functions. The former takes the union of encoded atoms in `ci.1` with encoded atoms with flipped sign in `ci.2`, while the latter uncodes and separates atoms in pair components, according to their sign:

```
Definition c2p ci : pinterp :=
  [set encodega             ga | ga in ci.1 ] ∪
  [set (flipga \o encodega) ga | ga in ci.2 ].

Definition p2c pi : cinterp :=
  ([set uncodega ga | ga in pi &    flag_pgatom ga],
   [set uncodega ga | ga in pi & ~~ flag_pgatom ga]).
```

The *reflection* lemma characterizing `c2p` is expressed by the `ciP` lemma below. Its proof relies on the cancelation properties for ground atom encoding and uncoding, as well as on the reflection lemma for set image membership.

```
Lemma ciP pga ci :
  pga ∈ (c2p ci) = if flag_pgatom pga then (uncodega pga) ∈ ci.1
                                      else (uncodega pga) ∈ ci.2.
Proof.
apply/idP/idP.
  by rewrite !inE; case/orP; case/imsetP ⇒ [ga ga_in →] /=;
     rewrite ?encodegaK ?encodegafK.
by rewrite !inE; case: ifP⇒ hf ha; apply/orP; [left|right];
   apply/imsetP; ∃(uncodega pga); rewrite // ?uncodegaKD ?
    uncodegaKDn // inE hf.
Qed.
```

Also, we establish `c2p` is a *bijection*, by proving the corresponding cancelation lemmas.

```
(* p2c is left inverse of c2p *)
Lemma c2pK : cancel c2p p2c.
(* c2p is left inverse of p2c *)
Lemma p2cK : cancel p2c c2p.

Lemma c2p_bij : bijective c2p.
```

[4] i.e, a set of ground atoms with positive flags

```
Proof. exact: Bijective c2pK p2cK. Qed.
```

9.4.2. Satisfiability Preservation

We establish that the satisfiability of a DATALOG program with negation in an interpretation is invariant to the complementation of the interpretation and to the encoding of the program into a positive one.

Satisfiability and Complementation A key property of complementing an interpretation ci with respect to a stratum ss is that, if the *negative dependency condition* holds, then *program satisfiability is preserved by complementation*, as stated in the `ciC_prog_true` lemma:

```
Lemma ciC_prog_true ss ci pp :
  all (predC (negdep^~ p)) (enum ss) →
  P.prog_true pp (c2p ci) →
  P.prog_true pp (c2p (ciC ss ci)).
```

Conversely, for a complemented interpretation ci that is already well-complemented interpretation with respect to a stratum ss, if the *positive dependency condition* holds, then *program satisfiability is preserved by program uncoding and positive model projection*, as stated in the `sanitize_prog_true` lemma:

```
Lemma sanitize_prog_true ss ci pp :
  ci_wc ss ci → {in p, ∀ cl, posdep ss cl} →
  P.prog_true pp (c2p ci) →
  prog_true (uncodep p) ci.1.
```

Satisfiability and Encoding Having identified and stated the *well-complementation property* of cumulative interpretations, we can now prove that *program satisfiability is preserved by encoding*, as stated in the `prog_true_pos` lemma below.

Given a program p, a complemented interpretation ci and a set of symbols ss, if the positive component of ci is a model of p and – additionally, the program symbols are contained in the strata ss with the respect to which ci is well-complemented – then, the corresponding positive interpretation is a model of ⌜p⌝.

```
Lemma prog_true_pos p ci ss :
  prog_true p ci.1 →
  ci_wc ss ci → {subset sym_prog p ≤ ss} →
  P.prog_true (encodep p) (c2p ci).
```

9.4.3. Stratified Evaluation Algorithm

As we will see in the following, a crucial phase in the stratified evaluation of a program is constructing complemented intepretations for its slices. To this end, we need to keep track of the strata accumulated at each iteration. For convenience, we wrap `cinterp`, together with a set of symbols, into a new type, `sinterp`, for *cumulative intepretations*.

```
Definition sinterp := (cinterp * {set symtype})%type.
```

Given a stratifiable program `p` and a stratification `str`, the `eval_prog` evaluation algorithm below computes a model for `p`, stratum by stratum. At each step, we run the positive engine `pengine_step` on the (positive encoding of the) program slice corresponding to the current stratum `encodep (slice_prog p (si.2 ∪ ss))`, seeded with the complemented model of the previous strata (see Section 9.4.1) `si.1`.

Note that the positive engine is defined in the same way as in Section 8.6.2, i.e, as iterating the forward chain operator as many times as there are elements in the program bound `bp`. The only difference is that we have to transform the resulting positive interpretation into a complemented one.

```
Definition bp : pinterp := setT.

Definition pengine_step pdef pp ci0 :=
  p2c (iter #|bp| (P.fwd_chain pdef pp) (c2p ci0)).
```

The positive engine will then add the facts that can be inferred for the current strata. However, it will not *explicitly* add the "complemented facts", i.e those corresponding to negated ground atoms that *implicitly* hold, by not being in the intepretation (see Section 9.1.2). To this extent, `eval_prog` must *complement* the model, as done by applying the `ciC` function. Once we have a complemented model for the current stratum, we process the rest through a recursive call.

```
Implicit Types (str : strata) (si : sinterp).

Variables (pdef : constant) (p : program) (p_sf : prog_safe p).

Fixpoint eval_prog str si : sinterp :=
match str with
  | [::]     ⇒ si
  | ss :: ps ⇒
    let p_curr  := slice_prog p (si.2 ∪ ss) in
    let m_next  := pengine_step pdef (encodep p_curr) si.1 in
    let m_compl := ciC ss m_next in
    eval_prog ps (m_compl, si.2 ∪ ss)
```

```
end.
```

9.5. Stratified Evaluation Properties

In Section 9.5.1, we outline additional characteristic properties required for the reuse of the "positive" inference engine from Section 8. Also, in Section 9.5.2, we state the identified stratified evaluation invariant, capturing characteristic properties of our extended, "negative" inference engine.

9.5.1. Properties of "Positive" Program Evaluation

As detailed next, the following properties of positive program evaluation were identified as relevant in our development: *soundness and boundedness*, *subsumption*, *stability*, *stratifiability*, *positivity*, *injectivity*, *modularity* and *incrementality*.

Soundness and Boundedness The main characterization of the positive engine evaluation, as described in Section 8, is captured by the `pengine_trueP` reflection lemma below. As such, in the presence of a *safe* positive program `pp`, the evaluation of `pp` – given an initial interpretation `ci0` – is *sound* and *bound* by `bp`, the Herbrand Base of `pp`.

```
Lemma pengine_trueP pdef pp ci0 (p_safe: P.prog_safe pp) :
  [∧ P.prog_true pp (c2p (pengine_step pdef pp ci0))
   &  c2p (pengine_step pdef pp ci0) ⊆ bp].
```

The proofs for the next three properties follow directly from the cancelation lemma for complemented interpretations and from the analogous results in Section 8.

Subsumption The evaluation of any positive program, given an arbitrary initial complemented interpretation, contains the latter.

```
Lemma pengine_subset pdef pp ci :
  c2p ci ⊆ c2p (pengine_step pdef pp ci).
  Proof. by rewrite p2cK; exact: P.iter_fwd_chain_subset. Qed.
```

Stability Given any safe positive program `pp` and an arbitrary complemented interpretation `ci` that satisfies it, its evaluation given `ci` adds no further facts.

```
Lemma pengine_stable pdef pp ci
  (h_sf : P.prog_safe pp)
```

```
  (h_tr : P.prog_true pp (c2p ci)) :
  pengine_step pdef pp ci = ci.
Proof. by rewrite /pengine_step P.iter_fwd_chain_stable ?c2pK. Qed.
```

(Symbol) Stratifiability Given a positive program `pp` and an initial complemented interpretation `ci`, (ground) atoms in the output of evaluating `pp` given `ci` are either *initial*, i.e from `ci`, or *derived*, i.e their symbols appear in the heads of clauses in `pp`.

```
Lemma pengine_sym pdef pp ci pga :
  (pga ∈ c2p (pengine_step pdef pp ci)) →
  (pga ∈ c2p ci) || (sym_gatom pga ∈ P.hsym_prog pp).
Proof. by rewrite p2cK; move/P.iter_fwd_chain_sym. Qed.
```

Positivity The evaluation of any positive program `pp`, given an arbitrary initial complemented interpretation `ci`, only outputs positive facts. In other words, the second "negative" component of `ci` is invariant with respect to evaluation.

```
Lemma pengine_idN pdef pp ci : (pengine_step pdef pp ci).2 = ci.2.
```

From the *subsumption* property above, a sufficient condition for establishing the result is proving `(pengine_step def pp ci).2 ⊆ ci.2`. Indeed, we show that `ga ∈ ci.2`, for any `ga`, such that `ga ∈ (pengine_step def pp ci).2`. The latter is equivalent to `ga` belonging to the set of uncoded negative atoms in the outcome of iterating the positive engine #|`bp`| times, which, by `ciP`, is the same as `flipga (encodega ga) ∈ c2p (pengine_step def pp ci)`. From the *stratifiability* property above, it follows that `ga` is either in `ci.2` or has a symbol with a negative flag among the head symbols of `pp`, which is clearly false, since negated atoms cannot appear in head clauses.

Injectivity Given two extensionally equal positive programs, their evaluations – given an arbitrary initial interpretation `ci` – are equal. The proof is by induction on the number of iterations of the positive engine evaluation, using the analogous property of the forward chaining algorithm from Section 8.

```
Lemma eq_pengine_step pdef pp1 pp2 ci (eq_p : pp1 =i pp2) :
  pengine_step pdef pp1 ci = pengine_step pdef pp2 ci.
```

Modularity Assume a safe positive program `pp1` and a positive program `pp2`, whose head symbols do not appear among the symbols of `pp1`. Starting from a model of `pp1`, i.e, `pi`, the evaluation of their concatenation equals the union of their respective evaluations.

```
Lemma pengine_cat pdef pp1 pp2 pi
      (h_ss  : [disjoint P.hsym_prog pp2 & P.sym_prog pp1])
      (h_sfb : P.prog_safe pp1)
      (h_tr  : P.prog_true pp1 pi) :
 pengine_step pdef (pp1 ++ pp2) (p2c pi) =
 pengine_step pdef pp1 (p2c pi) ∪ pengine_step pdef pp2 pi.
```

Incrementality The final result is the *incrementality property* of the positive evaluation of an encoded sliced program, given a well-complemented initial interpretation. This lemma is *fundamental* in establishing the strong soundness and completeness result for the normal program evaluation engine (see Section 9.6.1), hence we expand on it.

Theorem 9.5.1 (Incrementality of Positive Program Evaluation)**.** *Let* p *be a stratifiable program,* $\mathrm{si} = (\mathrm{ci},\ \mathrm{str}_{\leq})$ *a cumulative interpretation of* $\mathrm{p}_{\mathrm{str}_{\leq}}$ *and* ss *a current stratum. Assume that:*

- H1 $\equiv \ulcorner \mathrm{p}_{\mathrm{str}_{\leq}} \urcorner$ *symbols are not in* $\ulcorner \mathrm{p}_{\mathrm{ss}} \urcorner$ *heads, i.e,* $\mathrm{sym} \ulcorner \mathrm{p}_{\mathrm{str}_{\leq}} \urcorner \cap \mathrm{hsym} \ulcorner \mathrm{p}_{\mathrm{ss}} \urcorner = \emptyset$
- H2 $\equiv \mathrm{p}_{\mathrm{str}_{\leq}}$ *symbols are contained in* $\mathrm{str}_{\leq}$*, i.e,* $\mathrm{sym}\ \mathrm{p}_{\mathrm{str}_{\leq}} \subseteq \mathrm{str}_{\leq}$
- H3 $\equiv$ ci *is* well-complemented *with respect to* $\mathrm{str}_{\leq}$*, i.e,* $\mathrm{wc}_{\mathrm{str}_{\leq}}\mathrm{ci}$
- H4 $\equiv$ *the positive component of* ci satisfies $\mathrm{p}_{\mathrm{str}_{\leq}}$*, i.e,* $\mathrm{ci.1} \models \mathrm{p}_{\mathrm{str}_{\leq}}$

Then, the (positive) evaluation of $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \urcorner$ increments ci*, the input complemented interpretation corresponding to* $\mathrm{p}_{\mathrm{str}_{\leq}}$*, with a set of* positive facts i_{ss} *with symbols in* ss.

This is encoded as:

```
Lemma ci_decomposition ss si :
  [disjoint P.hsym_prog (encodep (slice_prog p ss))
   & P.sym_prog (encodep (slice_prog p si.2))] →
  {subset sym_prog (slice_prog p si.2) ≤ si.2} →
  ci_wc si.2 si.1 → prog_true (slice_prog p si.2) si.1.1 →
  (∃ i_ss,
     pengine_step pdef
       (encodep (slice_prog p (ss ∪ si.2))) si.1 =
       (si.1.1 ∪ i_ss, si.1.2)
     & {subset isyms i_ss ≤ ss}).
```

Proof. The set of positive facts i_{ss} equals `(pengine_step` $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \urcorner$ `ci).1 \ ci.1`. First, we show that, indeed, `(pengine_step` $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \urcorner$ `ci)` equals

`(ci.1 ∪ (pengine_step` $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \urcorner$ `ci).1 \ ci.1, ci.2).`

This holds as, according to `pengine_idN`, `(pengine_step` $\ulcorner p_{str_\leq \cup ss} \urcorner$ `ci).2 = ci.2`.

The second part of the proof is more intricate; namely, proving that the symbols in i_{ss}, i.e the symbols in `(pengine_step` $\ulcorner p_{str_\leq \cup ss} \urcorner$ `ci).1` $\setminus$ `ci.1`, are entirely contained in ss. We show that, for any ground atom `ga`, where `ga` $\in$ `(pengine_step` $\ulcorner p_{str_\leq \cup ss} \urcorner$ `ci).1` and `ga` $\notin$ `ci.1`, its symbol belongs to ss. To this end, we have to separate the evaluation of the program slicing with the current stratum ss from that of the program slicing with inferior strata symbols, i.e, $str_\leq$. As intermediate results we establish:

1. Ht $\equiv$ `c2p ci` $\models \ulcorner p_{str_\leq} \urcorner$, from H2, H3, H4 and the *preservation of satisfiability with respect to encoding* (see Section 9.4.2)
2. Hs $\equiv$ `P.prog_safe` $\ulcorner p_{str_\leq \cup ss} \urcorner$, from the *invariance of safety with respect to slicing and encoding* (see Section 9.1.1)
3. Hd $\equiv \ulcorner p_{str_\leq \cup ss} \urcorner = \ulcorner p_{str_\leq} \urcorner \cup \ulcorner p_{ss} \urcorner$, from *slicing decomposition* (see Section 9.3.3) and *encoding injectivity* (see Section 9.2.1)

From Ht, from program evaluation *injectivity* and, by using H1, Ht and Hs, from *modularity*, we can derive that `pengine_step` $\ulcorner p_{str_\leq \cup ss} \urcorner$ `ci` is equal to

$$\texttt{pengine\_step} \ulcorner p_{str_\leq} \urcorner \texttt{ ci} \cup \texttt{pengine\_step} \ulcorner p_{ss} \urcorner \texttt{ ci}.$$

Also, from Ht and Hs, via the *stability* property, we have:

$$\texttt{pengine\_step} \ulcorner p_{str_\leq} \urcorner \texttt{ ci} = \texttt{ci}.$$

Hence, `ga` $\in$ `(pengine_step` $\ulcorner p_{str_\leq \cup ss} \urcorner$ `ci).1` is equivalent to `ga` $\in$ `ci.1` or `ga` $\in$ $\ulcorner p_{ss} \urcorner$`.1`, which only leaves the latter, since we know that `ga` $\notin$ `ci.1`. Proving that the symbol of `ga` is in ss, follows from `ga` $\in \ulcorner p_{ss} \urcorner$`.1`, as a consequence of the *symbol stratifiability property*. □

9.5.2. Properties of "Negative" Program Evaluation

Let p be a stratifiable program and si = (ci, $str_\leq$), a cumulative interpretation. This packs together the complemented interpretation ci of the program slice $p_{str_\leq}$ and the corresponding accumulated strata symbols $str_\leq$. The crucial properties of the emph-stratified evaluation of p - concerning *program slicings* and *complemented interpretations* - are captured by the below *invariant*:

- the "positive" component of ci is indeed a *model* for $p_{str_\leq}$, i.e, ci.1 $\models p_{str_\leq}$
- the set of symbols in $p_{str_\leq}$ is contained in $str_\leq$, i.e, sym $p_{str_\leq} \subseteq str_\leq$
- ci is *well-complemented* with respect to $str_\leq$, i.e, $wc_{str_\leq}$ ci
- the set of symbols in ci is contained in $str_\leq$, i.e, sym ci $\subseteq str_\leq$

```
Definition si_invariant si :=
  [∧ prog_true (slice_prog p si.2) (sanitize_model si.1)
   ,  sym_prog (slice_prog p si.2) ⊆ si.2
   ,  ci_wc si.2 si.1
   &  ci_syms si.1 ⊆ si.2].
```

As discussed in the chapter's introduction, we are interested in verifying that the previously described algorithm actually computes a model for a stratified program, as captured by the above invariant. The fully formal argument requires quite a bit of technical detail, as shown by the corresponding proof, overviewed next.

9.6. Characterization of the "Negative" Engine

In Section 9.6.1, we present the main proof of soundness, termination and completeness of the previously presented inference engine. Also, we discuss a proof of "relative" minimality for the model it computes (see Section 9.6.2).

The basic idea of the proof in Section 9.6.1 is to perform induction on the number of stratas. It is easy to see that, for a program with a single symbol stratum, the positive engine will output the right model. Thus, the interesting part of the proof is establishing that one step of `eval_prog` will produce a model for the current program, sliced up to the current strata. According to the main result from Section 5, in the first evaluation step we obtain a model for the encoded program. At this point, an obvious question, answered in the proof of Theorem 9.6.1 below, is:

When is a model of an encoded program also a model of the original one ?

Once we answer this, we prove complementation preserves truth. In fact, the strata condition is crucial here, as explicit complementation will add negated facts to our previous complemented model. However, given that no clause body can depend on such a fact, validity is not altered. Finally, we apply the induction hypothesis with our newly generated complemented model. The conditions for induction are indeed met, as the model is well-complemented by construction.

9.6.1. Strong Soundness-Completeness of Stratified Program Evaluation

Theorem 9.6.1. *Let* p *be a program,* str *a stratification of* p *- consisting of lower strata* $\mathrm{str}_{\leq}$ *and upper strata* $\mathrm{str}_{>}$[5] *- and* ci *a complemented interpretation. If the cumulative* input interpretation $(\mathrm{ci}, \mathrm{str}_{\leq})$ *satisfies the invariant conditions, then the* output interpretation *of the one-step evaluation of* $\mathrm{p}_{\mathrm{str}_{>}}$*, given* $(\mathrm{ci}, \mathrm{str}_{\leq})$*, also satisfies them.*

The above theorem is encoded as:

[5] i.e, $\mathrm{str}_{\leq}$ stratifies $\mathrm{p}_{\mathrm{str}_{\leq}}$ and $\mathrm{str}_{>}$ stratifies $\mathrm{p}_{\mathrm{str}_{>}}$

```
Lemma eval_prog_true (ci, str≤) str> :
is_strata_rec p str> str≤ → si_invariant (ci, str≤) →
si_invariant (eval_prog str> (ci, str≤)).
```

Proof. The proof follows by induction on $\texttt{str}_{>}$.

Base Case $\texttt{str}_{>} = \texttt{[::]}$. Conclusion trivially holds.

Step Case $\texttt{str}_{>} = \texttt{ss :: ps}$.

Induction Hypothesis Generalizing over the cumulative interpretation, we have that:

$$\texttt{IH: } \forall \texttt{(ci, str}_{\leq}\texttt{), is\_strata\_rec p ps str}_{\leq} \rightarrow \texttt{si\_invariant (ci, str}_{\leq}\texttt{)} \rightarrow \texttt{si\_invariant (eval\_prog ps (ci, str}_{\leq}\texttt{))}$$

From the theorem hypothesis we know:

- `is_strata_rec p (ss :: ps)` $\texttt{str}_{\leq}$, i.e according to Section 9.3.1
  - `str_ps` $\equiv$ `is_strata_rec p ps (`$\texttt{str}_{\leq}$ `∪ ss)`
  - `strI` $\equiv$ `[disjoint str`$_{\leq}$ `& ss]`
  - `strN` $\equiv$ `all (fun s ⇒ ∼∼ negdep s` $\mathrm{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$`) (enum ss)`
  - `strP` $\equiv$ `all (posdep str`$_{\leq}$ `∪ ss)` $\mathrm{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$
- `si_invariant si`, i.e according to Section 9.5.2
  - `si_true` $\equiv$ ci.1 $\models \mathrm{p}_{\texttt{str}_{\leq}}$
  - `si_sub` $\equiv$ sym $\mathrm{p}_{\texttt{str}_{\leq}} \subseteq \texttt{str}_{\leq}$
  - `si_wc` $\equiv$ $\mathrm{wc}_{\texttt{str}_{\leq}}$ ci
  - `si_str` $\equiv$ sym ci $\subseteq \texttt{str}_{\leq}$

Unfolding the goal `si_invariant (eval_prog ss :: ps (ci,` $\texttt{str}_{\leq}$`))`, we obtain: `si_invariant (eval_prog ps (`$\mathrm{C}_{\texttt{ss}}$ `(pengine_step` $\mathrm{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$ `ci),` $\texttt{str}_{\leq}$ `∪ ss))`, which we will prove from the induction hypothesis.

Applying `IH`, the new goals become:

- `is_strata_rec p ps` $\texttt{str}_{\leq}$ `∪ ss`
- `si_invariant (`$\mathrm{C}_{\texttt{ss}}$ `(pengine_step` $\mathrm{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$ `ci),` $\texttt{str}_{\leq}$ `∪ ss)`

The first goal is directly provable via `str_ps`, while the second amounts to showing:

- `G1` $\equiv$ `(`$\mathrm{C}_{\texttt{ss}}$ `(pengine_step` $\mathrm{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$ `ci)).1` $\models$ $\mathrm{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$

- `G2` $\equiv$ sym $\mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \subseteq \mathrm{str}_{\leq} \cup \mathrm{ss}$
- `G3` $\equiv \mathrm{wc}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}$ (C_{ss} `(pengine_step` $\mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}$ `ci))`
- `G4` $\equiv$ sym (C_{ss} `(pengine_step` $\mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}$ `ci))` $\subseteq \mathrm{str}_{\leq} \cup \mathrm{ss}$

Before proving the above, we establish a couple of needed intermediate results.

- `hD` $\equiv$ `[disjoint` hsym $\ulcorner \mathrm{p}_{\mathrm{ss}} \urcorner$ `&` sym $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq}} \urcorner$`]`.

  From `slice_progP`, it follows that hsym $\ulcorner \mathrm{p}_{\mathrm{ss}} \urcorner \subseteq$ ss. Since sym $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq}} \urcorner \subseteq \mathrm{str}_{\leq}$ (from `si_sub`) and `[disjoint` $\mathrm{str}_{\leq}$ `& ss]` (from `strI`), `hD` holds.
- `hP` $\equiv$ `{in` $\mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}$`, ∀ cl : clause, posep` ($\mathrm{str}_{\leq} \cup \mathrm{ss}$) `cl}`, implied by `strP`.

We start by proving `G3`, as it is necessary in deriving `G1` and `G4`.

Proof of `G3` From `hD`, `si_true`, `si_sub` and `si_wc`, using the *incrementality* of positive evaluation, i.e the `ci_decomposition` lemma (see Section 9.5.1), we have:

$$\texttt{pengine\_step}\ \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}\ \texttt{ci} = \texttt{(ci.1} \cup \texttt{i\_ss, ci.2)}$$

Thus, $\mathrm{wc}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}$ (C_{ss} `(pengine_step` $\mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}$ `ci))` is equivalent to:

$$\mathrm{wc}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}\ (\mathrm{C}_{\mathrm{ss}}\ \texttt{(ci.1} \cup \texttt{i\_ss, ci.2))}$$

which, from the definition of complementation (see Section 9.4.1), equals:

$$\mathrm{wc}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}\ \texttt{(ci.1} \cup \texttt{i\_ss, (ci.1} \cup \texttt{i\_ss)}^{\mathrm{C}_{\mathrm{ss}}} \cup \texttt{ci.2)}$$

From `strI` and `si_sub`, given the *well-complementation union decomposition property*, as stated in the `wcUP` lemma (see Section 9.4.1), the above goal can be split into:

- $\mathrm{wc}_{\mathrm{str}_{\leq}}$ `ci`, which is exactly `si_wc`
- $\mathrm{wc}_{\mathrm{ss}}$ `(i_ss, (ci.1` $\cup$ `i_ss)`$^{\mathrm{C}_{\mathrm{ss}}}$`)`, i.e
  - $\texttt{i\_ss}^{\mathrm{ss}} \cup \texttt{((ci.1} \cup \texttt{i\_ss)}^{\mathrm{C}_{\mathrm{ss}}}\texttt{)}^{\mathrm{ss}} = \top^{\mathrm{ss}}$
  - $\texttt{i\_ss}^{\mathrm{ss}} \cup \texttt{((ci.1} \cap \texttt{i\_ss)}^{\mathrm{C}_{\mathrm{ss}}}\texttt{)}^{\mathrm{ss}} = \emptyset$

  both of which are provable using properties of *interpretation and complemented interpretation slicing*, as overviewed in Section 9.3.3

Proof of `G1` As a consequence of *safety preservation with respect to slicing and encoding* (see Section 9.1.1), it holds that `P.prog_safe` $\ulcorner \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \urcorner$. Hence, according to the *soundness* of positive evaluation, i.e `pengine_trueP` (see Section 9.5.1):

$$\texttt{c2p (pengine\_step } \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \texttt{ ci)} \models \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}.$$

From this and `strN`, due to *satisfiability preservation with respect to complementation*:

$$\texttt{c2p (C}_{\mathrm{ss}} \texttt{ (pengine\_step } \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}} \texttt{ ci))} \models \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}.$$

Using the above result, together with `strP` and `G3`, we reach the needed conclusion, by applying the `sanitize_prog_true` lemma (see Section 9.4.1).

Proof of `G2` We show that, for an arbitrary cl, where $\mathrm{cl} \in \mathrm{p}_{\mathrm{str}_{\leq} \cup \mathrm{ss}}$, for any symbol s, $\mathrm{s} \in \mathrm{sym\ cl}$, it follows that $\mathrm{s} \in \mathrm{str}_{\leq} \cup \mathrm{ss}$. The proof is by case analysis on whether s is a *head symbol* or a *body symbol* of cl. In the first case the conclusion trivially holds, by the definition of program slicing. In the latter, the conclusion holds due to the `strP` *positive-dependency property* of the program stratification.

Proof of `G4` As before, using positive evaluation *incrementality* and unfolding the definition of complementation, the goal becomes:

$$\mathrm{sym}\ \texttt{(ci.1} \cup \texttt{ i\_ss, (ci.1} \cup \texttt{ i\_ss)}^{\mathrm{C_{ss}}} \cup \texttt{ ci.2)} \subseteq \mathrm{str}_{\leq} \cup \mathrm{ss}$$

which is equivalent to:

$$(\mathrm{sym}\ \texttt{ci}) \cup (\mathrm{sym}\ \texttt{i\_ss}) \cup (\mathrm{sym}\ \texttt{(ci.1} \cup \texttt{ i\_ss)}^{\mathrm{C_{ss}}}) \subseteq \mathrm{str}_{\leq} \cup \mathrm{ss}$$

and which, from the definitions of (complemented) interpretation slicing, reduces to $(\mathrm{sym}\ \texttt{ci}) \cup \mathrm{ss} \subseteq \mathrm{str}_{\leq} \cup \mathrm{ss}$. This follows from the `si_str` invariant property. □

9.6.2. Minimality and Uniqueness of the Computed Model

As a corrolary of Theorem 9.6.1, given a stratifiable DATALOG program with negation p, it follows that the encoded evaluation engine - as defined in Section 9.4.3 - indeed computes a *model* for p. However, a more subtle discussion concerns its *minimality* (and *uniqueness*), as illustrated by the example below.

Example 9.6.2. *Consider the program:*

$$P = \left\{ \begin{array}{l} p \leftarrow q \\ r \leftarrow \neg q \\ s \leftarrow \neg q \\ t \leftarrow \neg q \end{array} \right\}$$

stratifiable, according to $\sigma(p) = 1$ *and* $\sigma(q) = \sigma(r) = \sigma(s) = \sigma(t) = 2$, *into* $P = P_1 \sqcup P_2$, *where:* $P_1 = \left\{ \begin{array}{l} p \leftarrow q \end{array} \right\}$ *and* $P_2 = \left\{ \begin{array}{l} r \leftarrow \neg q \\ s \leftarrow \neg q \\ t \leftarrow \neg q \end{array} \right\}$.

According to the stratified semantics:

- $M_1 = T_{P_1} \uparrow \omega(\emptyset) = \emptyset$
- $M_2 = T_{P_2} \uparrow \omega(M_1) = \{r, s, t\}$

The computed model *is* $M_P = \{r, s, t\}$*, different from the model* $M_P^{min} = \{p\}$*, which is smaller than* M_P*.*

As discussed in Section 6, this is due to the *non-monotonicity* of the extended immediate consequence operator. From the stratification properties, we can treat *already derived facts* as part of the extensional database (i.e, *edb*) – since their veracity is *fixed* – and, consequently, later refer to them negatively. While the *edb* naturally expands during bottom-up evaluation, the set of *newly derived facts* can actually decrease. Hence, the minimality of a computed stratified model depends on *fixing* its input. Equivalently, a model is minimal with respect to others, if they *agree on the submodel relative to the accumulated stratification*. Since when expressing minimality we cannot talk about models globally, but need to consider previous ones, as well as current candidates, we state it independently from the strata invariant conditions (as defined in Section 9.3.1).

The condition is formalized recursively below. Let p be a program with accumulated strata $\text{str}_{\leq}$ and current stratum ss. For any well-complemented interpretation, whose positive component $\text{ci}_{\leq}.1$ is a model of $\ulcorner \text{p}_{\text{str}_{\leq}} \urcorner$, the *computed model* for $\ulcorner \text{p}_{\text{str}_{\leq} \cup \text{ss}} \urcorner$ given $\text{ci}_{\leq}$, i.e (pengine $\ulcorner \text{p}_{\text{str}_{\leq} \cup \text{ss}} \urcorner$ $\text{ci}_{\leq}$).1, is *minimal* with respect to all models of $\ulcorner \text{p}_{\text{str}_{\leq}} \urcorner$ containing $\text{ci}_{\leq}.1$.

```
Fixpoint is_min_str_rec p str> str≤ :=
  match str> with
  | [::]     ⇒ True
  | ss :: ps ⇒
    (∀ (ci≤ : cinterp),
        ci_wc str≤ ci≤ → ci_syms ci≤ ⊆ str≤ →
        prog_true (slice_prog p str≤) ci≤.1 →
        ∀ (i_ss : interp),
          i_ssym ss i_ss = i_ss →
          let p_next := slice_prog p (str≤ ∪ ss) in
          prog_true p_next (ci≤.1 ∪ i_ss) →
          (pengine_step pdef (encodep p_next) ci≤).1
          ⊆ (ci≤.1 ∪ i_ss))
    ∧ is_min_str_rec ps (str≤ ∪ ss)
  end.
```

The main result concerning model minimality in our setting establishes it as a consequence of the strata invariant, i.e as relative to stratification:

```
Lemma minimality p str> str≤ :
  is_strata_rec p str> str≤ → is_min_str_rec str> str≤.
```

Proof. The proof follows by induction on $\texttt{str}_{>}$.

Base Case $\texttt{str}_{>} = \texttt{[::]}$. The conclusion trivially holds.

Step Case $\texttt{str}_{>} = \texttt{ss :: ps}$.

Induction Hypothesis We have that:

$$\texttt{IH: } \forall\ \texttt{str}_{\leq}\texttt{, is\_strata\_rec p ps str}_{\leq} \rightarrow \texttt{is\_min\_str\_rec p ps str}_{\leq} \cup \texttt{ss}$$

From the theorem hypothesis we know:

- `is_strata_rec p (ss :: ps)` $\texttt{str}_{\leq}$, i.e according to Section 9.3.1
  - `str_ps` $\equiv$ `is_strata_rec p ps (`$\texttt{str}_{\leq} \cup$ `ss)`
  - `strI` $\equiv$ `[disjoint str`$_{\leq}$ `& ss]`
  - `strN` $\equiv$ `all (fun s` $\Rightarrow \sim\sim$ `negdep s` $\texttt{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$`) (enum ss)`
  - `strP` $\equiv$ `all (posdep str`$_{\leq} \cup$ `ss)` $\texttt{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$
- fixing a cumulative interpretation $\texttt{ci}_{\leq}$
  - `h_wc` $\equiv \texttt{wc}_{\texttt{str}_{\leq}}$ ci
  - `h_true` $\equiv \texttt{ci}_{\leq}\texttt{.1} \models \texttt{p}_{\texttt{str}_{\leq}}$

We prove that `(pengine` $\ulcorner \texttt{p}_{\texttt{str}_{\leq} \cup \texttt{ss}} \urcorner$ $\texttt{ci}_{\leq}\texttt{).1} \subseteq \texttt{ci}_{\leq}\texttt{.1} \cup \texttt{i\_ss}$, for any addition `i_ss` of current stratum facts to $\texttt{ci}_{\leq}\texttt{.1}$, such that satisfiability is preserved, i.e

$$\texttt{ci}_{\leq}\texttt{.1} \cup \texttt{i\_ss} \models \texttt{p}_{\texttt{str}_{\leq} \cup \texttt{ss}}$$

The proof is based on using the above condition, together with the minimality of computed positive program models:

```
Definition is_min_pmodel pi pp ci0 :=
  ∀pi', (c2p ci0) ⊆ pi' → P.prog_true pp pi' → pi ⊆ pi'.
```

Indeed, the only remaining obligation becomes:

$$\texttt{c2p ci}_{\leq} \subseteq \texttt{c2p (ci}_{\leq}\texttt{.1} \cup \texttt{i\_ss, ci}_{\leq}\texttt{.1} \cup \texttt{i\_ss}^{\texttt{C}_{\texttt{ss}}}\texttt{)}$$

which follows from the monotonicity of the `c2p` transformation and from properties of (complemented) intepretation slicing. □

9.7. Discussion

9.7.1. Contributions

In this chapter we presented a Coq/ssreflect formalization of the theory of Datalog with negation, based on the Mathematical Components library for sets over finite types, i.e, `finset`. The formalization consists of the following components:

- *mechanization of the syntax and semantics of Datalog programs with negation*
- *mechanization of stratified evaluation*:
  we formalize program stratification and slicing of programs and interpretations; in order to reuse the positive engine, we translate negated literals to flagged positive atoms and extend the notion of an interpretation to that of a "complemented interpretation";
- *extending the theory of the positive engine*:
  a crucial part of the stratified evaluation relies on the positive engine to perform evaluation of negative programs encoded as positive; this reuse requires an extension to the theory of the positive engine with incrementality and modularity lemmas
- *formal characterization of the negative engine*:
  the key properties we establish are soundness, termination and completeness and model minimality

9.7.2. Lessons

What are the technical challenges in establishing the key characterization theorem ?

The central problem is the large amount of housekeeping needed, since every single object in the main proof is defined relative to a set of strata symbols. We believe we could improve our representation and data structures to ease this task; however, this comes at a non-trivial cost. Another source of intricacy is the large amount of side-conditions occuring in the proof. In addition to safety and other proof obligations already present in the positive case, here the strata condition is composed of four separate predicates, as well as an induction invariant. We do not think this can be easily improved, as these conditions are intrinsic to the proof. Indeed, we believe that the length of the main proof, even if a bit dense, is reasonable at approx. 60 loc.

Quoting [1]: "no precise characterization of stratified semantics in model-theoretic terms has emerged". To conclude, we regard the formal proofs given in this section as a step towards addressing this concern.

Part IV.

Evaluation

10. Conclusion

In this chapter, we evaluate the formal development presented previously, with regard to proof effort, reusability and scalability (see Section 10.1) and by summarizing the learned lessons (see Section 10.2).

10.1. Evaluation

Proof Effort We detail the proof effort that went into the formalization presented in Chapter 8 and Chapter 9. Note that we heavily used SSReflect features and proofs natively provided by the Mathematical Components library. As such, we estimate the development to be particularly compact relative to the comprehensive nature of the results established. As a comparison, the Coq formalization of standard Datalog given in [84] amounts to 2500 loc. This only contains a soundness proof for bottom-up inference, together with that of the procedure's decidability (a proof we did not have to carry out, as our development, by relying on small-step reflection, is computation-based and exploits the inherent decidability of booleans).

Our mechanization of standard Datalog programs over finite domains, of bottom-up inference, and of corresponding proofs for soundness, termination, completeness and computed model minimality, as well as of relevant incrementally proofs - needed for the extension of Datalog with negation -, amounts to approx. 1300 loc. An alternative development, supporting Datalog programs over infinite domains and establishing the same results - without the incrementally properties, as we did not base further work on the development[1] -, amounts to approx 1200 loc. The mechanization of Datalog with negation, stratified evaluation, together with corresponding proofs for soundness, termination, completeness and "relative" minimality of the computed model, amounts to approx. 1700 loc. Another (preliminary) development that explicitly analyzes programs - computing symbol arities, the active domain and building corresponding instances - and that establishes preservation of soundness results from the Datalog with negation, currently amounts to approx. 800. loc.

Reusability and Scalability We found that libraries and proofs developed in this setting are highly reusable. This observation is based, on the one hand, on the seemless integration of external proofs like those concerning fixpoint theory (see [33]) and, on the other hand, on the robust reuse of the library we developed for standard Datalog in

[1] indeed basing the library for negation in the infinite domain setting was not needed, as we show we can restrict ourselves to the active domain (see Section 8.1.1)

building that for its extension with negation.

Concerning scalability, we plan to further refine our development on several accounts. One example would be with respect to our library for DATALOG with negation. In the stratified evaluation, by indexing interpretations with the set of already processed symbols, we would inherently assure, at every step, that the Herbrand base of programs forms a well-complemented lattice. This would save us from having to explicity check this property holds. Related to this point, we would like to develop a library for lattice theory that would be particulary useful as basis for further language extensions, e.g, when introducing function symbols. Also, we would like to implement support for other types of semantics and evaluations, as presented in Section 5 and Section 6. Ideally, such an effort would ultimately result in providing a heuristic-agnostic evaluation procedure, together with corresponding, formally-checked, properties.

10.2. Lessons Learned

The exercise of formalizing database aspects has been an edifying experience. It helped clarify both the fundamentals underlying theoretical results and the proof-engineering implications of making these machine readable and user reusable.

On the database side, it quickly became apparent that, while foundational theorems appeared intuitively clear, if not obvious, understanding their *rigorous* justification required deeper reasoning. Resorting to standard references (even comprehensive ones, such as [1]), led at times to the realization that low-level details were either glanced over or left to the reader.

For instance, to the best of the author's knowledge, no *scrupulous* proofs of the soundness and completeness of DATALOG's forward-chain evaluation (as presented in Chapter 8 and Chapter 9) exist in the literature. Indeed, as these results are theoretically uncontroversial, their proofs are largely taken for granted and, understandably so, as they ultimately target database practitioners. Consequently, these are mostly *assumed* in textbook presentations or when discussing proofs in more complex settings, e.g, when studying further language extensions. It was only by constructing these proofs "from the ground up", in a proof assistant, that the relevance of different properties (e.g safety, finiteness, domain independence), the basis behind introducing certain definitions (e.g, predicate intensionality/extensionality, strata restrictions, logical consequence, stratified evaluation), or the precise meaning of ad-hoc notions and notations (e.g "substitution compatibility", $\mathbf{B}(\mathbf{P})$, model restrictions) became apparent.

As it is well known, database theory is based on solid mathematical foundations, from model theory to algebra. This suggests that, when compared to off-the-shelf program verification, verification in the database context requires that proof systems have good support for doing mathematics. It was an interesting lesson to discover, in practice,

the extent to which database theory proofs could be recast into matemathical ones. To exemplify, by expressing forward chain as an elegant set construct, we transferred proofs about Datalog inference engines into set-theory proofs, which are more natural to manipulate. Conversely, when formalizing the stratified semantics of Datalog with negation, we were compelled to resort to some ad-hoc solutions to handle the lack of native library support for lattice theory. Indeed, textbooks largely omit explainations as to why and how it is necessary to reason about such structures when proving properties of stratified evaluation. To this end, we were led to introduce specialized notions, such as interpretation complementation. Also, we had to *explicitly* establish that, at each evaluation step, the Herbrand base of the program's restriction with respect to the set of already processed strata symbols was a *well-complemented* lattice.

On the theorem proving side, a crucial lesson is the importance of relying on infrastructure that is well-tailored to the nature of the development. This emerged as *essential* while working on the formalization of standard Datalog. The triggering realization was that, as we could - without loss of generality - restrict ourselves to the active domain (see Section 8.1.1), models could be reduced to the finite setting and atoms could be framed as finite types (see Section 8.1.3). Therefore, the Mathematical Components library, prominently used in carrying out finite model theory proofs, stood out as best suited for our purposes. Indeed, since we could heavily rely on the convenient properties of finite types and on already established set theory properties, proofs were rendered much easier and more compact.

Apart from having good library support, making adequate choices for the type encoding of various constructs proved essential. Having experimented with many alternatives, we noticed first-hand the dramatic effect this could have on the size and complexity of proofs. For example, while having too many primitives is undesirable in programming language design, it turned out to be beneficial to opt for greater base granularity when formalizing Datalog. Separating the type of ground constructs from that of constructs with variables helped both at a conceptual level, in understanding the relevance of standard range restrictions, and at a practical one, in facilitating proof advancement. Another example concerns the mechanization of substitutions. Having the option to representing them as finitely supported functions, together with all the useful properties this type has, was instrumental to finding a suitable phrasing of the matching algorithm's soundness and completeness properties. Indeed, as the algorithm incrementally constructed groundings, it seemed natural to want to define an ordering on substitutions leading up to these. Being able to have a type encoding allowing to regard substitutions both as functions and as lists was essential for this purpose. A final example regards the formalization of models. As previously mentioned, setting up the type of ground atoms as finite payed off in that we could use many results and properties from the `fintype` library, when reasoning about models - which was often the case. In particular, we took advantage of the inherent lattice structure of such types.

The final lesson is that proof style is consequential not only a posteriori, for the sake

of readability or reusability, but also as an a priori way of approaching a formalization endeavour. In the early stages of getting used to working with a prover, it was very tempting to try and build proofs in a top-down manner, i.e, to primarily work on refining the goal itself to match the hypothesis. This does not seem to correspond to the natural way one would typically do proofs on paper and would lead to a feeling that the proof was driven by the machine and not the user. While this sometimes luckily works, it is largely up to the user to take the lead and direct the proof. As such, in the author's experience, it helps to not only do the detailed proof on paper beforehand, but also to proceed in a bottom-up manner, i.e, to rely on stating intermediate assertions based on the hypothesis. Lastly, relying on characteristic properties (SSReflect's so called P-lemmas), many of which are conveniently stated as reflection lemmas, led to leaner proofs by compositionality. In cases in which induction would have been the default approach, these provided a shorter alternative (also, see [38], in which a comprehensive fomalization of linear algebra is developed with *no induction*).

11. Perspectives

We consider that our formalization work opens many perspectives, both targeting language extensions (see Section 11.1) and different domains of application (see Section 11.2).

11.1. Language Extensions

Many additions to the standard DATALOG language have been proposed in the literature over the years. We have so far only applied theorem techniques to DATALOG with negation. A logical step would then be to mechanize further such extensions and to provide formal proofs regarding their semantics. Such an effort would make precise the impact that the interaction between different operators, e.g, $\exists$, $\vee$ and update constructs, could have on language properties.

11.1.1. Datalog with Existentials

As the best known and most widely used rule-based language, DATALOG has application in fields ranging from Data Integration and Exchange to Artificial Inteligence, i.e, the Semantic Web. However, additions to the standard DATALOG language are generally required. In particular, these concern enabling support for expressing local-as-view constraints or for ontology reasoning relying on description logics, whose concept axioms may include the existential restriction of concepts. Hence, we envision formalizing the theory DATALOG$^{\exists}$, a highly expressive languge extension allowing for existentially quantified variables in rule heads. The main challenge in this context is the undecidability of query answering. It would be interesting to mechanize inference and optimization techniques for restricted fragments based on different paradigms, such as guardedness, weak-acyclicity, stickiness or shiness. An important step would be formally studying terminating versions of the chase procedure, such as the Parsimonious Chase ([52]), or the application of rewriting algorithms, such as Magic Sets ([8]).

11.1.2. Datalog with Disjunction

A natural extension to the language of DATALOG with negation, presented in Section 9, is Disjunctive DATALOG. This allows for disjunction of atoms in the head of rules and can be refined to also support negation in rule bodies. In this setting, one line of work could be formally establishing precise semantics for the language, based on standard approaches in disjunctive logic programming. Also, another aspect could concern the formal analysis of such programs, in particular of properties, such as stratification, modularity and rewritability.

11.1.3. Datalog with Updates

In order to make Datalog more amenable to real-world system integration, another extension regards the support of dynamic behaviour. This translates to incorporating update constructs and to providing adequate incremental maintanence algorithms, as an alternative to materialization. Given the declarative nature of the language, this setting is a particularly favorable one for analyzing problems that notoriously occur when dealing with updates, such as the view-update problem. As such, other lines of work could be proposing a mechanized semantics in the presence of insertion and deletion and studying relevant language restrictions for the unambiguous evaluation of rules with updated heads.

11.2. Applications

We also outline two potential applications that would make use of our work in formalizing Datalog languages and corresponding inference engines.

11.2.1. Enforcing Security

Another perspective of our work is to extend Datalog to a security language, capable of expressing fine-grained access control rules and delegation operations. These could then be used to define security policies, whose enforcement can be formally established by providing corresponding Coq proofs. The development of such a framework would be targeted to Data Integration and Exchange problems. The main goal would be to ensure strong privacy guarantees, while working with heterogeneous data sources and supporting data sharing.

11.2.2. System Certification

The renewed interest in the Datalog language has begun to feel its presence in industry as well. A prominent example is the LogicBlox commercial Datalog system ([43], [4]). This constitutes a promising opportunity to integrate formal certification techniques with real-word database system. As such, one research direction would be to first adapt and extend our the Coq specification of our inference engine, to account for the wide range of queries supported by the LogicBlox one (in line with the language extensions mentioned above). Then, we could ideally employ property-based testing tools, such as QuickChick [31], to ensure that outputs conform to consistency constraints formulated as invariants. This is particularly interesting in the context of generally undecidable inference procedures, such as the chase. As QuickChick has only recently been incorporated into the SSReflect proving process, by exploiting refinement methodology proposed in [30] and [24], this endeavour would be good occasion to assess the feasability of employing such a workflow.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995. ISBN 0-201-53771-0. URL `http://webdam.inria.fr/Alice/`.

[2] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. *The Theory of Joins in Relational Databases*. *ACM Trans. Database Syst.*, 4(3):297–314, September 1979. ISSN 0362-5915. doi: 10.1145/320083.320091. URL `http://doi.acm.org/10.1145/320083.320091`.

[3] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0-934613-40-0. URL `http://dl.acm.org/citation.cfm?id=61352.61354`.

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. *Design and Implementation of the LogicBlox System*. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1371–1382, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742796. URL `http://doi.acm.org/10.1145/2723372.2742796`.

[5] William Armstrong. *Dependency Structures of Database Relationships*. In *IFIP Congress*, pages 580–583, 1974.

[6] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. *The Matita Interactive Theorem Prover*. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23: 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 64–69. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22438-6. doi: 10.1007/978-3-642-22438-6_7. URL `http://dx.doi.org/10.1007/978-3-642-22438-6_7`.

[7] Leo Bachmair and Harald Ganzinger. *Resolution Theorem Proving*. I:19–99, 2001.

[8] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. *Magic Sets and Other Strange Ways to Implement Logic Programs*. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, pages 1–15, 1986. doi: 10.1145/6012.15399. URL `http://doi.acm.org/10.1145/6012.15399`.

[9] Pablo Barceló and Reinhard Pichler, editors. *Datalog 2.0'12: Proceedings of the Second International Conference on Datalog in Academia and Industry*, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-32924-1.

[10] Henk P. Barendregt. *Handbook of Logic in Computer Science*. volume II, chapter *Lambda Calculi with Types*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992. ISBN 0-19-853761-1. URL `http://dl.acm.org/citation.cfm?id=162552.162561`.

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Z. Béguelin. *Formal Certification of Code-based Cryptographic Proofs*. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101, 2009. doi: 10.1145/1480881.1480894. URL `http://doi.acm.org/10.1145/1480881.1480894`.

[12] Catriel Beeri and Moshe Y. Vardi. *The Implication Problem for Data Dependencies*. In Shimon Even and Oded Kariv, editors, *ICALP: Annual International Colloquium on Automata, Languages and Programming*, pages 73–85. Springer Berlin Heidelberg, 1981. ISBN 978-3-540-38745-9. doi: 10.1007/3-540-10843-2_7. URL `http://dx.doi.org/10.1007/3-540-10843-2_7`.

[13] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. *A Relational Library*, 2013. URL `http://datacert.lri.fr/esop/html/Datacert.AdditionalMaterial.html`.

[14] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. *A Coq Formalization of the Relational Data Model*. In Zhong Shao, editor, *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 189–208, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54833-8. doi: 10.1007/978-3-642-54833-8_11. URL `http://dx.doi.org/10.1007/978-3-642-54833-8_11`.

[15] Yves Bertot, Pierre Castéran, Gérard Huet, and Christine Paulin-Mohring. *Interactive Theorem Proving and Program Development : Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, Berlin, New York, 2004. ISBN 978-3-540-20854-9. URL `http://opac.inria.fr/record=b1101046`. Données complémentaires: http://coq.inria.fr.

[16] Sandrine Blazy and Xavier Leroy. *Formal Verification of a Memory Model for C-like Imperative Languages*. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005. URL `http://gallium.inria.fr/~xleroy/publi/memory-model.pdf`.

[17] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. *Formal Verification of a C Compiler Front-End*. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006. URL `http://gallium.inria.fr/~xleroy/publi/cfront.pdf`.

[18] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. *A Trusted Mechanised JavaScript Specification*. pages 87–100, 2014. doi: 10.1145/2535838.2535876. URL `http://doi.acm.org/10.1145/2535838.2535876`.

[19] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *What You Always Wanted to Know About Datalog (And Never Dared to Ask). IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, mar 1989. ISSN 1041-4347. doi: 10.1109/69.43410. URL `http://dx.doi.org/10.1109/69.43410`.

[20] Ashok K. Chandra and Philip M. Merlin. *Optimal Implementation of Conjunctive Queries in Relational Databases*. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 77–90, New York, NY, USA, 1977. ACM. doi: 10.1145/800105.803397. URL `http://doi.acm.org/10.1145/800105.803397`.

[21] Chin-Liang Chang and Richard C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science Classics. Academic Press, 1973. ISBN 978-0-12-170350-9.

[22] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. *Effective Interactive Proofs for Higher-Order Imperative Programs*. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596565. URL `http://doi.acm.org/10.1145/1596550.1596565`.

[23] Edgar F. Codd. *A Relational Model of Data for Large Shared Data Banks. Communications of the ACM*, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL `http://doi.acm.org/10.1145/362384.362685`.

[24] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. *Refinements for Free!* In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs: Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, pages 147–162, Cham, 2013. Springer International Publishing. ISBN 978-3-319-03545-1. doi: 10.1007/978-3-319-03545-1_10. URL `http://dx.doi.org/10.1007/978-3-319-03545-1_10`.

[25] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development*

System. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.

[26] Evelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. *Mechanically Proving Termination Using Polynomial Interpretations. Journal of Automated Reasoning*, 34(4):325–363, 2005. doi: 10.1007/s10817-005-9022-x. URL `http://dx.doi.org/10.1007/s10817-005-9022-x`.

[27] Haskell B. Curry and Robert Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

[28] Dirk Van Dalen. *Logic and Structure*. Springer-Verlag, 1983.

[29] Nicolaas G. de Bruijn. *AUTOMATH, A Language for Mathematics*. T.H., Department of Mathematics, Eindhoven University of Technology, November 1968.

[30] Maxime Dénès, Anders Mörtberg, and Vincent Siles. *A Refinement-Based Approach to Computational Algebra in Coq*. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving: Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 83–98, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32347-8. doi: 10.1007/978-3-642-32347-8_7. URL `http://dx.doi.org/10.1007/978-3-642-32347-8_7`.

[31] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. QuickChick: Property-based testing for Coq (abstract). In *VSL*, 2014. URL `http://www.easychair.org/smart-program/VSL2014/index.html`.

[32] Robert A. Di Paola. *The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas. J. ACM*, 16(2):324–327, 1969. ISSN 0004-5411. doi: 10.1145/321510.321524. URL `http://doi.acm.org/10.1145/321510.321524`.

[33] Christian Doczkal and Gert Smolka. *Completeness and Decidability Results for CTL in Coq*. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 226–241, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08970-6. doi: 10.1007/978-3-319-08970-6_15. URL `http://dx.doi.org/10.1007/978-3-319-08970-6_15`.

[34] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. *Specifying and Reasoning About Dynamic Access-Control Policies*. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 632–646, 2006. doi: 10.1007/11814771_51. URL `http://dx.doi.org/10.1007/11814771_51`.

[35] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. *Packaging Mathematical Structures.* In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 327–342, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9. doi: 10.1007/978-3-642-03359-9_23. URL http://dx.doi.org/10.1007/978-3-642-03359-9_23.

[36] Michael Gelfond and Vladimir Lifschitz. *The Stable Model Semantics For Logic Programming.* pages 1070–1080. MIT Press, 1988.

[37] Georges Gonthier. *The Four Colour Theorem: Engineering of a Formal Proof.* In *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, page 333, 2007. doi: 10.1007/978-3-540-87827-8_28. URL http://dx.doi.org/10.1007/978-3-540-87827-8_28.

[38] Georges Gonthier. Point-free, set-free concrete linear algebra. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving: Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, pages 103–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-22863-6. doi: 10.1007/978-3-642-22863-6_10. URL http://dx.doi.org/10.1007/978-3-642-22863-6_10.

[39] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. *A Machine-Checked Proof of the Odd Order Theorem.* In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013. doi: 10.1007/978-3-642-39634-2_14. URL http://dx.doi.org/10.1007/978-3-642-39634-2_14.

[40] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system.* Research Report RR-6455, Inria Saclay Ile de France, 2015. URL https://hal.inria.fr/inria-00258384.

[41] Carlos Gonzalía. *Relations in Dependent Type Theory.* PhD thesis, Chalmers Göteborg University, 2006.

[42] Claude C. Green and Bertram Raphael. *Research on Intelligent Question Answering Systems.* Technical report, 1968.

[43] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. *LogicBlox, Platform and Language: A Tutorial.* pages 1–8, 2012. doi: 10.1007/978-3-642-32925-8_1. URL http://dx.doi.org/10.1007/978-3-642-32925-8_1.

[44] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. *CertiKOS: A Certified Kernel for Secure Cloud Computing*. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 3:1–3:5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1179-3. doi: 10.1145/2103799.2103803. URL `http://doi.acm.org/10.1145/2103799.2103803`.

[45] John Harrison. *The HOL Light Theory of Euclidean Space. Journal of Automated Reasoning*, 50(2):173–190, 2013. doi: 10.1007/s10817-012-9250-9. URL `http://dx.doi.org/10.1007/s10817-012-9250-9`.

[46] Michael Hedberg. *A Coherence Theorem for Martin-LöF's Type Theory. Journal of Functional Programming*, 8(4):413–436, July 1998. ISSN 0956-7968. doi: 10.1017/S0956796898003153. URL `http://dx.doi.org/10.1017/S0956796898003153`.

[47] William A. Howard. *The Formulas-as-Types Notion of Construction*. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.

[48] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. *Datalog and Emerging Applications: An Interactive Tutorial*. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989456. URL `http://doi.acm.org/10.1145/1989323.1989456`.

[49] Jael Kriener, Andy King, and Sandrine Blazy. *Proofs You Can Believe In. Proving Equivalences Between Prolog Semantics in Coq*. In Tom Schrijvers, editor, *15th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 37–48, Madrid, Spain, September 2013. ACM. URL `https://hal.inria.fr/hal-00908848`.

[50] J. Kuhns. *Answering Questions by Computer: A Logical Study*. Technical report, 1967.

[51] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. *CakeML: A Verified Implementation of ML*. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, 2014. doi: 10.1145/2535838.2535841. URL `http://doi.acm.org/10.1145/2535838.2535841`.

[52] Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. *Efficiently Computable Datalog∃ Programs*. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*, 2012. URL `http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4521`.

[53] Xavier Leroy. *Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant*. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006. URL `http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf`.

[54] Xavier Leroy. *Formal Verification of a Realistic Compiler*. *Communications of ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL `http://doi.acm.org/10.1145/1538788.1538814`.

[55] Pierre Letouzey. *A Library for Finite Sets*.

[56] Roger Levien and M. E. Maron. *Relational Data File: A Tool for Mechanized Inference Execution and Data Retrieval*. Technical report, 1965.

[57] Leonid Libkin. *The Finite Model Theory Toolbox of a Database Theoretician*. In *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009, June 19 - July 1, 2009, Providence, Rhode Island, USA*, pages 65–76, 2009. doi: 10.1145/1559795.1559807. URL `http://doi.acm.org/10.1145/1559795.1559807`.

[58] John W. Lloyd. *Foundations of Logic Programming*. 1984.

[59] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. *Testing Implications of Data Dependencies*. *ACM Transactions on Database Systems*, 4(4):455–469, December 1979. ISSN 0362-5915. doi: 10.1145/320107.320115. URL `http://doi.acm.org/10.1145/320107.320115`.

[60] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. *Toward a Verified Relational Database Management System*. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2010, pages 237–248, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706329. URL `http://doi.acm.org/10.1145/1706299.1706329`.

[61] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2016. URL `https://coq.inria.fr/refman/`. Version 8.5pl2.

[62] Robin Milner. *Logic for Computable Functions: Description of a Machine Implementation*. 1972.

[63] Jack Minker. *Perspectives in Deductive Databases*. *Journal of Logic Programming*, 5 (1):33–60, March 1988. ISSN 0743-1066. doi: 10.1016/0743-1066(88)90006-4. URL `http://dx.doi.org/10.1016/0743-1066(88)90006-4`.

[64] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. *Ynot: dependent types for imperative programs*. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240, 2008. doi: 10.1145/1411204.1411237. URL `http://doi.acm.org/10.1145/1411204.1411237`.

[65] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL : A Proof Assistant for Higher-Order Logic*. volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9. URL http://dx.doi.org/10.1007/3-540-45949-9.

[66] Ulf Norell. *Dependently Typed Programming in Agda*. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, pages 230–266, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04652-0. doi: 10.1007/978-3-642-04652-0_5. URL http://dx.doi.org/10.1007/978-3-642-04652-0_5.

[67] Adrian Onet. The Chase Procedure and its Applications in Data Exchange. PhD thesis, 2012.

[68] Sam Owre, John M. Rushby, and Natarajan Shankar. *PVS: A Prototype Verification System*. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, pages 748–752, 1992. doi: 10.1007/3-540-55602-8_217. URL http://dx.doi.org/10.1007/3-540-55602-8_217.

[69] Bill Parducci. *Extensible Access Control Markup Language (XACML) Specification*, 2005.

[70] Christine Paulin-Mohring. *Inductive Definitions in Type Theory*. Accreditation to supervise research, Université Claude Bernard - Lyon I, December 1996. URL https://tel.archives-ouvertes.fr/tel-00431817.

[71] T. C. Przymusinski. *Foundations of Deductive Databases and Logic Programming*. chapter *On the Declarative Semantics of Deductive Databases and Logic Programs*, pages 193–216. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. ISBN 0-934613-40-0. URL http://dl.acm.org/citation.cfm?id=61352.61357.

[72] T. C. Przymusinski. *Well-Founded Semantics Coincides with Three-Valued Stable Semantics*. *Fundamenta Informaticae*, 13:445–463, 1990.

[73] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (3. ed.)*. McGraw-Hill, 2003. ISBN 978-0-07-115110-8.

[74] Raymond Reiter. *Logic and Databases*. chapter *Deductive Question-Answering on Relational Databases*, pages 149–177. Springer US, Boston, MA, 1978. ISBN 978-1-4684-3384-5. doi: 10.1007/978-1-4684-3384-5_6. URL http://dx.doi.org/10.1007/978-1-4684-3384-5_6.

[75] John Alan Robinson. *Automatic Deduction with Hyper-Resolution*. *International Journal of Computer Mathematics*, I:27–234, 1965.

[76] John Alan Robinson and Andrei Voronkov, editors. Handbook of Automated Reasoning. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001. ISBN 0-444-50812-0.

[77] Oded Shmueli. *Equivalence of Datalog Queries is Undecidable. Journal of Logic Programming*, 15(3):231–241, February 1993. ISSN 0743-1066. doi: 10.1016/0743-1066(93)90040-N. URL `http://dx.doi.org/10.1016/0743-1066(93)90040-N`.

[78] Alfred Tarski. *A Lattice-Theoretical Fixpoint Theorem and its Applications. Pacific Journal of Mathematics*, 5(2):285–309, 1955. URL `http://projecteuclid.org/euclid.pjm/1103044538`.

[79] Jeffrey D. Ullman. Principles of Database and Knowledge-base Systems, Volume I. Computer Science Press, Inc., New York, NY, USA, 1988. ISBN 0-88175-188-X.

[80] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. 1990.

[81] M. H. Van Emden and R. A. Kowalski. *The Semantics of Predicate Logic as a Programming Language. J. ACM*, 23(4):733–742, October 1976. ISSN 0004-5411. doi: 10.1145/321978.321991. URL `http://doi.acm.org/10.1145/321978.321991`.

[82] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. *The Well-founded Semantics for General Logic Programs. Journal of the ACM*, 38(3):619–649, July 1991. ISSN 0004-5411. doi: 10.1145/116825.116838. URL `http://doi.acm.org/10.1145/116825.116838`.

[83] Nathan Whitehead. A certified distributed security logic for authorizing code. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs*, TYPES'06, pages 253–268, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74463-0, 978-3-540-74463-4. URL `http://dl.acm.org/citation.cfm?id=1789277.1789294`.

[84] Nathan Whitehead, Jordan Johnson, and Martín Abadi. *Policies and Proofs for Code Auditing*. In *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*, ATVA'07, pages 1–14, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-75595-0, 978-3-540-75595-1. URL `http://dl.acm.org/citation.cfm?id=1779046.1779048`.

[85] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. *Formalizing the LLVM Intermediate Representation for Verified Program Transformations. SIGPLAN Notifications*, 47(1):427–440, January 2012. ISSN 0362-1340. doi: 10.1145/2103621.2103709. URL `http://doi.acm.org/10.1145/2103621.2103709`.

Synthèse en français

Dans le contexte actuel de la prolifération rapide des volumes de données, l'obtention de garanties fortes sur la fiabilité des systèmes et des applications qui traitent ces données massives est essentiel. À cette fin, une piste consiste à appliquer des techniques déjà bien connues dans le domaine de la vérification et de la preuve de programmes. En particulier, le recours à des assistants à la preuve tels que Coq, pour formaliser les systèmes et certifier formellement les algorithmes, est une voie très prometteuse.

Dans cette thèse nous nous intéressons à fournir des spécifications formelles provenant de deux approches différentes pour la définition des modèles de données. L'une est basée sur les relations mathématiques - c'est-à-dire le *modèle relationnel* - et l'autre est basée sur la logique - c'est-à-dire le *modèle déductif* - représenté par le langage Datalog. Le *modèle relationnel* permet de *représenter* l'information à travers des relations, d'*affiner* l'information ainsi représentée en la restreignant par des contraintes d'intégrité et d'*extraire* cette information grâce à des langages de requêtes. Ces derniers sont fondés soit sur l'algèbre relationnelle, soit sur le calcul relationnel.

Deux versions équivalentes du modèle relationel co-existent : la version par positions et la version par noms. Dans le cas par positions, les attributs spécifiques à une relation sont ignorés, et seule l'arité (c'est-à-dire le nombre de ces attributs) est utilisée par les langages de requêtes. Dans le cas nommé, au contraire, les attributs font partie intégrante de la base et sont utilisés tant par les langages de requêtes que pour définir les contraintes d'intégrité.

En pratique, les systèmes tels qu'Oracle, DB2, PostgreSQL ou encore Microsoft Access reposent sur la version *nommée* du modèle. Ce choix est motivé par plusieurs raisons. Les noms portent davantage de sens que des numéros, ce qui est appréciable à des fins de modélisation. De plus, les optimiseurs tirent parti des structures de données auxiliaires à des fins d'optimisation physique. De telles structures, en particulier les index, sont définies en utilisant explicitement les noms des attributs. C'est pourquoi nous avons choisi de formaliser cette version du modèle.

Le *modèle déductif* est issu du cel relationnel et de la programmation logique et étend l'éxpresivité du prémier avec le support pour la recursion et pour le calcul des clôtures transitives. En s'appuyant sur la logique, il fourni un cadre unificateur qui permet d'*extraire* non seulement des données explicitement stockées, mais aussi des données qu'on peut inférer.

Contributions

Les contributions techniques de cette thèse consistent en deux développements formels : une bibliothèque Coq pour le modèle relationnel et une bibliothèque Coq/ssreflect pour la programmation logique dans le fragment Datalog. Nous les détaillons comme suit.

- une bibliothèque Coq pour le modèle relationnel qui contient :
  - *une formalisation du modèle de données* (relations, tuples, etc.)
  - *une mécanisation des contraintes d'intégrité* :
    nous formalisons les dépendances générales et nous prouvons la correction de leur procédure d'inférence ; aussi, nous formalisons des instances correspondant aux deux

sous-classes de dépendances générales : les dépendances fonctionnelles et les dépendances multivaluées ; nous établissons la correction de leur inférence, ainsi que la complétude, dans le cas des dépendances fonctionnelles

  - *une mécanisation des principaux langages de requêtes relationnelles* :
    nous formalisons l'algèbre relationnelle et un fragment restreint de calcul relationnel, c'est-à-dire, les requêtes conjonctives
  - *les preuves des principaux "théorèmes de base de données"* :
    les équivalences algébriques, le théorème d'homomorphisme et la minimisation des requêtes conjonctives

- une mécanisation d'un moteur d'inférence pour les programmes Datalog positifs

  - *en utilisant le support* ssreflect *pour les types finis* :
    nous simplifions l'effort de vérification considérablement, en supposant, sans perte de généralité, la finitude des modèles ; en effet chaque programme Datalog a un modèle fini, pour cela, considérer le cadre fini suffit. Nous effectuons cette réduction séparément du développement principal, ce qui, à notre avis, simplifie la présentation du développement.
  - *une mécanisation évolutive de la syntaxe et de la sémantique de Datalog positif*
  - *mécanisation de l'heuristique d'évaluation bottom-up* :
    formalisation d'un algorithme itératif de matching monadique pour les termes, atomes et corps de clause, avec des preuves de correction et de complétude correspondantes
  - *caractérisation formelle du moteur positif* :
    les propriétés clés que nous établissons sont la correction, la terminaison, la complétude et la minimalité du modèle. Elles sont basées sur des preuves que nous donnons pour montrer que l'opérateur de conséquence immédiate est monotone, borné et stable, ainsi que sur des résultats de la théorie des points fixes.

- une formalisation ssreflect d'un moteur d'inférence pour des programmes Datalog avec négation

  - *mécanisation de la syntaxe et de la sémantique des programmes Datalog avec négation*
  - *mécanisation de l'évaluation stratifiée* :
    nous formalisons la stratification des prédicats et le découpage des programmes et des interprétations ; afin de réutiliser le moteur positif, nous traduisons les littéraux niés vers des atomes positifs avec des markeurs booléens et nous étondrons la notion d'interprétation à celle "d'interprétation complémentée" ;
  - *extension de la théorie du moteur positif* :
    une partie cruciale de l'évaluation stratifiée s'appuie sur le moteur positif pour effectuer l'évaluation de l'encodage de programmes négatifs codés comme positifs ; cette réutilisation nécessite une extension de la théorie du moteur positif avec des lemmes d'incrémentalité et de modularité
  - *caractérisation formelle du moteur négatif* :
    les propriétés clés que nous établissons sont la correction, la terminaison, la complétude et la minimalité du modèle

Formalisation du modèle relationnel

Dans cette partie, nous présentons une formalisation du modèle relationnel. Le développement comprend la partie définition de données, ainsi que les contraintes d'intégrité et leur inférence. Dans le cas des dépendances générales, nous formalisons la procédure du "chase" pour laquelle nous établissons la correction. Dans le cas plus restreint des dépendances fonctionnelles, nous formalisons le système d'Armstrong pour lequel nous établissons la correction et la complétude. De plus, nous formalisons l'algèbre relationnelle et les requêtes conjonctives et nous prouvons les équivalences algébriques, le théorème d'homomorphisme et la minimisation des requêtes conjonctives. Pour établir formellement les preuves de ces résultats, on a dû rendre explicites des notions techniques comme la fraîcheur des variables, l'unification et l'échappement de capture des variables. Ces aspects ne sont pas mentionnés dans la littérature et n'apparaissent pas dans des les implémentations correspondantes (le plus souvant écrites en C). Pourtant, le vrai défi est la modélisation des données, en particulier celle des attributs, tuples et relations. En faisant des choix opportuns, nous arrivons à rester générique et fidèle aux formalisations sur papier, existant dans les ouvrages de référence.

Formalisation du langage Datalog standard

Dans cette partie, nous présentons une formalisation du langage Datalog standard. La librairie contient une mécanisation du langage, de sa sémantique par théorie des modèles et par point fixe, ainsi qu'un moteur d'inférence implémentant l'évaluation bottom-up, c'est-à-dire le forward-chain. Le développement comprend des preuves de propriétés caractéristiques de tous les composants que nous avons défini. Le moteur d'inférence est la composante principale du développement. L'idée principale derrière son fonctionnement est comme suit. Essentiellement, le moteur itère un opérateur par point fixe, basée sur l'implémentation de la "conséquence immédiate" à travers un algorithme de matching. Le but de cette procédure d'inférence est la construction d'un modèle (minimal) pour un programme Datalog standard donné. À cette fin, le moteur maintient une interprétation (le modèle candidat) qu'il essaie d'améliorer de manière itérative. Pour ce faire, il identifie d'abord les clauses insatisfiables, c'est-a-dire celles pour lesquelles le corps de la clause est satisfait par l'interprétation actuelle, mais la tête de la clause ne l'est pas. L'interprétation actuelle est ensuite "réparée" en lui ajoutant les faits manquants qui correspondent à des têtes de clause saturées. Cette opération est réalisée grâce à un algorithme du forward-chain qui calcule les substitutions faisant le matching entre les atomes dans le corps de la clause et les faits de l'interprétation. La condition de sûreté du langage Datalog assure que toutes les variables présentes dans la tête de la clause sont aussi parmi celles dans le corps de la clause. Par conséquent, la substitution obtenue est close et son application à la tête de clause produit un nouveau fait. Une fois que l'interprétation actuelle est mise à jour avec tous les faits qu'on puisse inférer en une étape du forward-chain, la procédure est répétée jusqu'à l'obtention d'un point fixe. Nous montrons formellement que l'interprétation finale est un modèle minimal du programme de départ.

Formalisation du langage Datalog stratifié

Dans cette partie, nous présentons une formalisation du langage Datalog stratifié, basé sur le développement précédent pour le langage Datalog standard. La librairie contient une mécanisation du langage, de sa sémantique et d'un moteur d'inférence qui implémente l'évaluation stratifiée. Elle comprend des preuves des propriétés caractéristiques de tous les composants que nous avons défini. Le moteur d'inférence est la composante principale du développement

et nous donnons l'idée principale derrière son fonctionnement comme suit. Les programmes logiques avec négation sont plus difficiles à analyser que ceux sans négation, car, en particulier, l'existence d'un modèle (unique) minimale n'est pas toujours garantie. Pour remédier à cette situation, une condition suffisante est fournie dans la littérature : la stratification des prédicats. L'idée principale de cette technique est de découper le programme de sorte que chaque partie puisse être correctement traité comme s'il s'agissait d'un programme positif. Par conséquent, une telle partie a un modèle minimal qui peut être enrichi, de manière itérative, au fur et à mesure qu'on considère les strates suivants. Il y a des programmes Datalog qui ne sont pas stratifiables, mais, pour ceux qui le sont, une stratification peut être définie en calculant les dépendances entre les prédicats du programme. Il suit que le strate inférieur est une partie indépendante, sans négation, du programme original. Grâce au fait que la stratification assure l'indépendance des parties suivantes du programme, nous évaluons chacune par la sémantique du point fixe. La formalisation est impactée par la stratification de deux façons : premièrement, les modèles calculés sont indexés par des strates et, deuxièmement, chaque partie du programme avec négation peut être encodé comme un programme positif.