UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

A COQ FORMALIZATION OF RDF AND ITS APPLICATIONS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

TOMÁS JAVIER VALLEJOS PARADA

Este trabajo ha sido parcialmente financiado por ANID Programa ICM-ANID Proyecto ICN17_002N y ANID Fondecyt Proyecto Regular 1190058

SANTIAGO DE CHILE
2023

Resumen

Una formalización en Coq de RDF y sus aplicaciones

El Marco de Descripción de Recursos (del inglés Resource Description Framework, RDF) es un modelo de datos basado en grafos que enriquece los documentos de la World Wide Web con datos entendibles por máquinas. Durante sus más de dos décadas de desarrollo, RDF se ha adaptado a los nuevos desafíos tecnológicos. Esta adaptación de RDF ha permitido su aplicación en contextos críticos donde se necesitan garantías de seguridad, como calcular sumas de verificación o firmar digitalmente grafos RDF.

Los asistentes de prueba son herramientas de verificación formal capaces de proveer certificaciones de software sólidas, proporcionando un puente entre la programación y la lógica. En particular, Coq ha influido en los métodos formales, la verificación de programas, y matemáticas formales mediante el desarrollo de software confiable utilizando argumentos lógicos rigurosos. Visualizamos un estándar RDF con software certificado siguiendo un enfoque mecanizado mediante el uso de asistentes de prueba.

En este trabajo presentamos `CoqRDF`: una librería mecanizada que especifica formalmente el modelo RDF, proveyendo una base de datos de lemas formalmente mecanizados sobre RDF. `CoqRDF` nos permite definir operaciones como el re-etiquetado de nodos en blanco e isomorfismo RDF y razonar sobre ellos mecánicamente. Yendo más lejos, utilizamos la librería para implementar el algoritmo κ-*mapping*, un algoritmo para detectar grafos RDF que son isomorfos. Desarrollamos una técnica de prueba para trazabilizar el re-etiquetado de nodos en blanco durante la computación. Usando esta técnica y `CoqRDF` demostramos dos propiedades sobre κ-*mapping*: que para cualquier grafo, κ-*mapping* retorna un grafo isomorfo al grafo de entrada; y que dos grafos resultan en conjuntos iguales al aplicarles κ-*mapping*, si y solo si, los grafos de entrada son isomorfos. Estas pruebas nos permiten concluir que κ-*mapping* es un algoritmo iso-canonico con una prueba completamente mecanizada. Una perspectiva posible para `CoqRDF` sería mecanizar otros estándares construidos sobre RDF. Creemos haber contribuido con una primera piedra hacia una cadena de herramientas mecanizada de estándares web basada en RDF.

Abstract

The Resource Description Framework (RDF) is a graph data model enriching the documents of the World Wide Web with machine-readable data. For over two decades of development, RDF has been adapted to the new technological challenges. This adaptation of RDF has enabled its application in critical contexts where security guarantees are needed, such as computing checksums or digitally signing RDF graphs.

Proof assistants are formal verification tools allowing a robust certification of software, providing a bridge between programming and logic. In particular, Coq has influenced formal methods, program verification, and formal mathematics through the development of reliable software using rigorous logical arguments. We envision an RDF standard with certified software by following a mechanized approach using proof assistants.

In this work we present `CoqRDF`: a mechanized library formally specifying the RDF model, with a database of formally mechanized lemmas about RDF. `CoqRDF` allows us to define operations such as blank node relabeling, and RDF isomorphism and reason about them mechanically. Going further, we use the library to implement κ-mapping, an algorithm for detecting isomorphic RDF graphs. We develop a proof technique to keep track of blank node relabeling through computation. Using this technique and `CoqRDF` we prove two properties of κ-mapping: that κ-mapping returns a graph, which is isomorphic to input; and that two graphs give a set-equal result under κ-mapping, if and only if, the input graphs are isomorphic. These two properties allows us to conclude that κ-mapping is an iso-canonical algorithm, with a fully mechanized proof. A possible perspective for `CoqRDF` would be to mechanize standards built on top of RDF. We believe that we have contributed with a first stone towards a mechanized toolchain of web standards based on RDF.

A Francisca, Vivian y Ricardo.

Agradecimientos

A mi familia, por su paciencia; por su cariño; y por su apoyo incondicional ¡Sobre todo durante la escritura de este documento! Forjaron mi hambre de saber, y alimentaron sin descanso mi curiosidad; a ustedes debo mi inagotable entusiasmo por descubrir este mundo.

A Emilia, por su tenacidad para soportarme, su poderoso acompañar en los días menos felices, y su misteriosa habilidad para elevar el ánimo.

A Aidan y Éric, por presentarme esta intersección entre la web semántica y formalización. Gracias por supervisarme estos años, por estrujar mis esfuerzos, introducirme a la investigación, y enseñarme a calmar la frustración de perseguir caminos que avanzan, pero en la dirección incorrecta.

Thank you Assia, for trusting in this project and agreeing to guide me, for training me in the art of reasoning constructively, and for advising the confluence of each of the experiments we discussed.

A Pablo Barceló, Federico Olmedo, y Matías Toro, por aceptar ser parte de mi comisión y por permitirme perfeccionar este documento a través de sus comentarios.

A los mismísimos Enzo, Felipe e Ignacio, por cada una de las experiencias que nos unen desde el comienzo, por escuchar como centinelas sobre los asistentes de prueba, los grafos y como establecer un isomorfismo, pero brevemente, por acogerme cálidamente como su amigo.

A mis compañeres de oficina: a Tomás, por compartir el gusto por ssreflect. A Mara, por sus consejos categóricos. A Damián, por no cansarse de escuchar argumentos sobre lo interesantes que son los grafos.

A la comunidad Pleiad, por haberme recibido con los brazos abiertos y complementar tan harmónicamente el trabajo riguroso con la sana convivencia.

A Koen, por insistirme en hacer las pruebas con papel y lápiz.

Al Departamento de Ciencias de la Computación de la Universidad de Chile, el Instituto Milenio Fundamento de los Datos, y toda su gente.

Y a los amigos que hice en este viaje, con mención especial a Victor, Coon, y el extenso y diverso grupo de les Toquiamigues.

Table of Content

List of Figures

List of source codes

Part I

Introduction

Chapter 1

Introduction

The current Web has its foundations in documents as every website on the Web has a document as its core. This approach is friendly to humans, but not to machines that process this data. Search engines retrieve documents that match keyword queries, but data is not structured. Also, the amount of data available over time increases, with some of them describing the same resource. The Web, with a document centric organization, makes it difficult to query data spread over different documents. If someone needs to relate data within two or more sources, they will have to search for it and integrate it manually.

It is not difficult to think of scenarios where this challenge is met. For example: where was the X concept coined? One would have to look into the bibliography, and then, for every reference, look up if the concept was introduced before, or if the reference being checked introduces new references to check. Luckily, this data can be modeled as knowledge graphs, and queried with an expressive language, which can retrieve data spread over different sources without manual interaction. The information is on the Web, yet not on a single webpage where we could answer our question quickly. A lot of information is available on the Web, but finding answers to questions involving multiple web pages or websites requires a lot of work.

Besides that, on the Web, one can find data in various structured formats, each with their particular pros and cons. Depending on the format of a particular dataset, we may use different tools. Then, if we have to combine the results of multiple datasets, we have to transform the data from one representation to another. Such problems could be avoided by selecting one standard format for publishing and exchanging data on the Web [35].

The Web of Data aims to address this problem by enriching the Web with more and more content in increasingly machine-readable formats. The World Wide Web Consortium (W3C) thus proposed the Resource Description Framework (RDF) in 1999 [44]. RDF is the recommendation from the W3C to standardize the problem of publishing data of the World Wide Web. The Resource Description Framework (RDF) is widely used: $38 - 45\%$ of websites are using RDF syntaxes such as *JSON-LD* or *RDFa*. Other formats based on *RDFa* such as Open Graph are used in 64% of known websites [69].

The broad use of RDF pushes researchers to study its properties formally. With over two decades of RDF design, RDF has adapted to the newer technology challenges, modifying its specification [44, 11, 18]. Some of these decisions were never completely summarized in the RDF standards [71].

Proof assistants, such as Coq or Agda, are software tools that allow programmers to develop programs and state mathematical theorems about them. One can also develop formal proofs of these theorems and in that way certify program properties [70]. To enhance the verification of RDF we propose the first mechanized definition of the RDF 1.1 standard named `CoqRDF`. We define `CoqRDF` in the Coq proof assistant using the Mathematical Components libraries. `CoqRDF` allows any user to reason about the abstract syntax of RDF 1.1 and its operations.

At the time of this writing, a W3C working group is developing an extension of RDF 1.1 [31]. This working group has the mission to define a standard to uniquely and deterministically calculate a hash of RDF Datasets. Hence, we extend `CoqRDF` with formal specification of the canonicalization terminology [18, 34]. Notions such as RDF equality, RDF isomorphism, and iso-canonical algorithms are mechanically implemented in the `CoqRDF` library. We prove important properties of these relations, e.g. RDF isomorphism is an equivalence relation. We implement the κ-mapping algorithm, a base instance of an iso-canonical algorithm; and we then develop a technique to keep track of isomorphism through computation. This technique allows us to mechanically verify that κ-mapping is an iso-canonical algorithm. The use of RDF canonicalization involves critical applications, such as, the decentralized verification of authorship of data in the RDF format. We hope that `CoqRDF` serves as a baseline to keep developing results around RDF and its applications.

1.1 Goal

The goal of this thesis is to develop a library to reason formally about the RDF data model and its operations. This library has to:

- Adopt RDF 1.1 abstract syntax.
- Be mechanically implemented.
- Provide enough expressiveness to reason about complex RDF operations.

Some target operations to specify are blank node relabeling and RDF isomorphism [18], and also the notion of iso-canonical algorithms [34]. Another goal is to provide an infrastructure that allows us to prove that an implementation can meet and be correct with respect to such specifications.

1.2 Results

This thesis is in the formalization domain and addresses the RDF data model [18]. We design and develop a novel mechanized library: `CoqRDF`, to allow formal analysis and definitions on top of it. The formalization is implemented through the use of the Coq proof assistant [70, 17], and the ssreflect tactic language and methodology [26]. Throughout the formalization we use the Mathematical components' library [51].

`CoqRDF` accounts for the definitions of the W3C specification of the RDF model in its version 1.1 [18]. `CoqRDF` allows any user to reason about the RDF model with respect to terms, blank nodes, triples, and RDF graphs. For every RDF component we define a specification of its well-formedness, which we then prove that each component meets. We implement core RDF operations in the library such as comparison under set equality, relabeling, membership and isomorphism testing. We also prove that set-equality for RDF graphs and RDF isomorphism are an equivalence relation.

We then propose an application for the `CoqRDF` library. We use `CoqRDF` to formally specify the notion of iso-canonical algorithms, followed by an implementation of the κ-mapping algorithm [34], where we explain how to define the algorithm using functional programming. Subsequently, we develop a methodology to keep trace of isomorphism throughout computation, which allows us to prove that κ-mapping returns well-formed graphs; that κ-mapping returns graphs that are isomorphic to the input; and that two graphs give a set-equal result under κ-mapping, if and only if, the input graphs are isomorphic. The thesis ends with a discussion about the design choices we made, and summary of our contributions and perspectives for `CoqRDF`.

1.3 Overview

The document is structured to allow its gradual reading. Each part has a chapter about the theoretical background of the object under study, followed by the mechanized implementation of the `CoqRDF` library. Each chapter ends with a brief summary of the topics of the chapter. The structure of the remaining chapters is the following:

Part I: Introduction

Chapter 2: Background Presents the tools being used in this document. In particular: introduces the notion of interactive theorem provers; Coq the proof assistant; the ssreflect methodology and tactic language; and the Mathematical Components library. The background is presented as a tutorial.

Part II: Formalizing the RDF model

Chapter 3: The Resource Description Framework Presents the RDF data model, its abstract syntax, and how it has evolved over decades of development. It introduces the notion of terms, triples, RDF graphs, and operations over RDF graphs.

Chapter 4: `CoqRDF` : a mechanized definition of RDF Presents `CoqRDF`, a library we have developed to formally reason about the RDF model. Exposes how to model terms, triples, RDF graphs, and how these objects behave under relabeling.

Part III: An application: Reasoning about iso-canonical algorithms

Chapter 5: Isomorphism Presents a definition of graphs, and a notion of isomorphism. It extends this notion to introduce RDF isomorphism and its properties. It follows by extending `CoqRDF` with a formal definition of RDF isomorphism, by decomposing the relation in smaller components, which can be independently studied. Finally, it presents how to model RDF isomorphism in Coq, through the use of these smaller components.

Chapter 6: Iso-canonical algorithms Presents the notion of canonical instances, and iso-canonical algorithms, which then are formally mechanized in `CoqRDF`. Afterwards an implementation of the κ-mapping algorithm is provided. Subsequently, a technique is provided to keep track of isomorphism through computation, which enables a proof verifing that κ-mapping is an iso-canonical algorithm, first, that κ-mapping returns a graph, which is isomorphic to the input; and second, and that two graphs give a set-equal result under κ-mapping, if and only if, the input graphs are isomorphic.

Part IV: Discussion and conclusion

Chapter 7: Discussion Gives an overview of the process of mechanically formalizing the RDF model. It compares the different design choices made during the development, and evaluates the impact of using math-comp and ssreflect to define `CoqRDF`.

Chapter 8: Conclusion It summarizes the contributions of this thesis, and discusses the future work and perspectives for `CoqRDF`.

Chapter 2

Background

In this chapter we introduce background concepts for the work of this thesis. In Section 2.1 we explain the concept of interactive theorem provers. This is followed by a description of their features, properties and foundations as programming languages and logic systems. We also explain the decision to use the proof assistant Coq to implement our library.

Later, in Section 2.2, we present formalization projects using Coq about mathematics, and software verification. In Section 2.3, we describe some mechanically formalized projects related to data. We follow this with a very brief introduction to Coq's syntax. In the introduction we use the ssreflect tactic language and the Mathematical Components libraries in Section 2.4. We introduce the unary, sum and product types in Subsection 2.4.1, then, we follow the introduction presenting properties for natural numbers. Through these examples we then present how to adopt the small scale reflection methodology, and how to use the math-comp libraries in Subsections 2.4.3, 2.4.5, and 2.4.4.

2.1 Interactive Theorem Provers

When a human user interacts together with software to produce a formal proof we call that software an Interactive Theorem Prover (ITP). Although initially the work focused on automatic theorem proving, research has actively improved the support for human-guided proofs in formal systems [29].

Many ITPs encode their proofs exploiting the *Curry-Howard isomorphism* [38]. The Curry-Howard isomorphism establishes a correspondence between a given logic and a given programming language. This correspondence assigns: a type in the programming language for every proposition in the logic; a program with that type in the programming language for every proof of a proposition in the logic. Also, the Curry-Howard isomorphism links proof simplification with the evaluation of the related program [73].

There are different logic and programming language flavors when choosing an ITP. Some of them are: Isabelle, an ITP based on higher order logic; Agda, a dependently typed language that can be used as an ITP, based on Martin-Löf's intuitionistic type theory [53]; Lean [57] and Coq [70], based on the Calculus of Inductive Constructions [17]; and many others.

Throughout this thesis we use Coq the Proof Assistant. The reasons to choose it are multiple. For example, Coq was awarded the ACM Software System Award in 2013 [23] in recognition for its lasting influence in formal methods, programming languages, program verification, and formal mathematics. ACM highlighted its key enabling technology for certified software. Coq enables the extraction of the implementation to external programming languages like OCaml, Haskell or Scheme [46, 47].

Coq has one more big advantage. Using ITPs eases the trust process. We move the verification effort from humans to machines. We trust on a human check of the kernel, where all the verification effort goes, and to the theory behind the logic of the ITP. But, it does not stop there, Coq has managed to move its trusted-base from the code (trusted code base) to the meta-theory of the CIC (trusted theory base) by formalizing the kernel of Coq, in Coq. This was achieved by the MetaCoq project [65, 64], which built an implementation of the type checker of Coq's kernel, and proved it correct with respect to its formal specification. This does not contradict Gödel's second incompleteness theorem since part of the meta-theory was axiomatized. Hence, assuming the correctness of the specification they proved a correct implementation of Coq's kernel, making Coq to rely on its meta-theory rather than the code-base.

Comparatively, Coq has a long history as a proof assistant, being developed since the early 90's [17, 16, 58]. Coq has been used for a series of reliable software foundations [61, 60, 1], covering basic logic tools; the use of proof assistants to construct rigorous logical arguments, how to use functional programming as a bridge between programming and logic; the use of type systems for defining well-behavedness of programs in a given programming language, etc. Besides, Coq has proven to scale to projects with challenging architectures and software of great size [26, 45, 39]. All these reasons make us choose Coq to develop a library to formally reason about RDF.

2.2 Coq the Proof Assistant

As explained, Coq is an interactive theorem prover (ITP). It lets its users specify mathematical objects within a programming language called Gallina. Then, one can interactively generate machine checked proofs of mathematical statements. The proof checker is Coq's kernel; it validates that the user reasoned with correct arguments.

Coq is the option of choice when formally verifying software. Several areas have verified their results using it. Some of the most significant ones include a formally verified Proof of the Four Color Theorem by Georges Gonthier [24], a machine-checked constructive proof of the odd order theorem [25], and the Compcert project, a verified C compiler, which formalized almost all the C language (ISO C99) [45, 39].

As we introduced in the previous section, Coq's logic evolved from the Calculus of inductive constructions (CIC) [17]. As per many other proof assistants, it is based on the notion of propositions as types [38, 73]. This notion refers to a correspondence between mathematical propositions and types in programming languages. Therefore, one can use Coq for developing mathematical facts over programs. One can express the property "program p is correct" as a mathematical statement in Coq and then prove the statement.

Correctness is especially important in critical contexts, where the presence of bugs may have high impact consequences. In these scenarios confidence in programs has to be strong. ITPs can offer a great amount of confidence that a particular resource is bug free, or in other words, that the resource is correct with respect to its specification. Indeed, ITPs have been used in the past to formalize security properties. The first Common Criteria Evaluation Assurance Level 7 (cc EAL 7) certification in the world [13] used an ITP to develop their formal model and proofs.

2.3 Mechanized formalization of Data

Formal semantics are important when trying to reason about programs. Therefore, mechanized formalization efforts have started to emerge in other fields such as Databases. It is common to find natural language specifications defining the behavior of data management systems and languages, but the pursuit of more complex and robust results has led to a demand for higher level guarantees. In general, mechanized approaches are a method to provide a faithful reference implementation and provide a basis for further investigation with interactive mathematical tools.

Let us review some examples of formal verification relating to data.

Chu et al. [14], developed a machine-checkable denotational semantics for an SQL fragment in the Coq proof assistant. The fragment includes, `select from where` with aggregation, but without `having`. Afterwards they use it to formally validate SQL *rewrite rules*, and to implement an automated procedure for deciding the equivalence of conjunctive queries. Later, Chu et al. [15] propose the U-semiring structure to model a fragment of SQL semantics, which they defined using the Lean proof assistant. They define an axiomatic interpretation of key constraints and axiomatize the operations of U-semiring to prove equivalence of queries in the presence of integrity constraints. They describe how to encode SQL queries into the U-semiring structure, and give a sound definition of SQL expressions equivalence under U-semiring semantics. Then, they develop a decision procedure to test the equivalence of queries of a fragment of SQL, and use it to automatically prove the correctness of rewrite rules from the literature.

Benzaken et al. [8], used the Coq proof assistant to specify the low level layer of SQL's execution engines. They design a high-level specification for data-centric operators and provide two implementations: one for physical algebra and another for SQL algebra. They formally prove that these two algebras implement the same semantics, relating the semantics of the query execution plan and the algebraic expression resulting from the query's semantic analysis. Contributing towards a mechanized SQL compilation chain.

There are more works on SQL relational algebra, for example, the work of Benzaken and Contejean [7], which presented a mechanized definition of a realistic fragment of SQL;

consisting of `select [distinct] from where group by having` queries with `null` values, functions, aggregates, quantifiers and nested potentially correlated sub-queries. They used this definition to build a certified semantic analyzer for an SQL compiler using Coq's extraction mechanism. Afterwards, the semantic analyzer was related to a formalization of an extended relational algebra, proving equivalences upon which most compilation optimizations are based, yielding the first formal mechanized proof of the equivalence of SQL and extended relational algebra.

There are works regarding data provenance as well. Benzaken et al. [9] proposed a formalized provenance-aware extended relational algebra for a subset of database queries. They validate the formalization with an adequacy proof with respect to standard evaluation of queries. Leading towards a posteriori certification of provenance for data manipulation, which is formally verified. Such a guarantee is crucial in domains such as medicine, to prescribe treatments. Opening the door to the development of certified data provenance in scientific workflows with guaranteed reproducibility.

In 2020, Lai et al. [41] showed how standardized semantic web technologies can be reproduced in a unified manner. This manner was achieved by using Coq and dependent type theory. The result was a system to build and query dependently typed knowledge graphs. Another example of formal verification is the mechanized formalization of GraphQL presented in 2020 [20]. The formalization of GraphQL contributed with a baseline for proving robust formal results.

Other works have focused on practical issues, such as a better representation of temporal data [48]. In addition, works have applied type theory for representing ontological structures in Coq [19], while some have built a graph library covering various notions of graphs and properties, implemented in Coq using ssreflect [22]. In all the cited efforts Coq has been the preferred proof assistant.

2.4 The Mathematical Components Library

Mathematical Components [51] is a library for the Coq proof assistant. It comprises several topics including numbers, lists, algebra, fields, etc. This library was first developed as the machinery for the verification of the proof of the Four Colors theorem [24], and then it was reused to verify the proof of the Odd Order Theorem [25].

For the reader not too familiar with Coq, we are presenting a (very) brief introduction to its syntax and features. For a more in depth explanation we refer the reader to the ssreflect user manual [26] and Coq manual [70]. In this tutorial we are using Coq version 8.17 and mathcomp version 1.6. Whenever the reader sees a comment in a code section, it should be understood as Coq's output after executing the line before the comment.

2.4.1 Polymorphic Finite Data Types

To model complex structures in the functional programming paradigm, we require Data Types that allow us to mix the different components of such a structure. The base types that allow this are the unary type, the sum type and the product type. The unary type represents a type with a single inhabitant. The sum of types corresponds to the different alternatives to build a certain structure, and the product of types corresponds to an ordered pair, which allows us to pair terms of different types. We now present how to model them using inductive types. We choose this presentation just to illustrate the notions of the unit type, sum type, and product type. However, it is usually these base types the ones which are used to model the inductive types. An explanation of the fundamental principles of these base types is beyond the scope of this thesis, but can be found in books; for example in "Types and Programming Languages" by Pierce [59], and in "Practical Foundations for Programming Languages" by Harper [27]. We will limit ourselves to showing how the basic applications are used and how they can be defined. The way of modeling data in Coq is with inductive types. These types are defined with the homonymous keyword `Inductive`; using it, we can define the unary type in the following way:

```
Inductive unit : Type :=
| tt.
```

With this declaration we define a type, which has only one possible way to construct it: `tt`. A binary sum takes two types and has two constructors, representing the alternative of generating a term from the first type, or the second one. We define it inductively as follows:

```
Inductive sum (T1 T2 : Type) : Type :=
| Left : T1 -> (sum T1 T2)
| Right : T2 -> (sum T1 T2)
```

We note the sum of the types `A` and `B` with `A + B`. With sum types we can model the type of the booleans: `bool`. We can represent it with the sum of `unit` and `unit`, thus, we say that the value constructed with `Left` represents `true`, and the one constructed with `Right` represents `false`. In Coq every constructor of an inductive type is disjoint from the other ones; for example, every `sum` built with `Left` will be different to any sum built with `Right`. We introduce definitions into Coq using the keyword `Definition`. We generate definitions for true and false, with their respective constructors.

```
Definition bool : Type := sum unit unit.
Definition true : bool := Left tt.
Definition false : bool := Right tt.
```

As an example we will use booleans to define the logical *and* for booleans: `andb`. We proceed by case analysis using the `match` keyword, which allows us to compare `b1` with `true` and `false`. If `b1` matches any of those patterns, then Coq evaluates the expression at the right-hand of the arrow: =>.

```
Definition andb (b1 b2 : bool) : bool :=
match b1 with
| true => b2
| false => false
end.
```

We have defined the `bool` type using a sum of `unit` and `unit`, and an operation using them: `andb`. Now we define products. A product receives two types and two terms of these types. We can define it in the following way:

```
Inductive prod (T1 T2 : Type) : Type :=
| Pair : T1 -> T2 -> (prod T1 T2).
```

We note the product of the types `A` and `B` with `A * B`. With `prod` we can define bool pairs, for example.

```
Definition pair_of_bool : Type :=
  Pair bool bool : bool -> bool -> (prod bool bool).
```

Sometimes, to specify that we are referring to the product or sum of types, we do so by adding the postfix `%type`, With this we syntactically instruct Coq that the symbols $*$ and $+$ have to be interpreted in the scope of the types, and not in any other way. Other inductive types, such as natural numbers, also note operations with `_ + _` and `_ * _`.

Having defined the basis for modeling structures in functional programming, we now present how to define propositions about inductively defined types.

2.4.2 Defining properties on the natural numbers

One may inductively define the natural numbers as follows:

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
```

The definition of `nat` encodes the unary representation of natural numbers. The inductive `nat` is a `Type` with two constructors: `O` representing the zero, and `S`, which takes a natural number and returns its successor. Many operations can be defined recursively on the natural numbers, for example, addition. In Coq, recursive functions are declared using the `Fixpoint` keyword. But, as Coq is a proof assistant every function has to terminate. If divergent functions were allowed, we could easily break Coq's logic consistency; for example, by defining the following program:

```
Fixpoint contra (b : bool) : False := contra b.
```

To prevent the introduction of logical inconsistency, Coq restricts recursive definitions. Recursive functions may only be applied to structurally smaller arguments. This restriction is a syntactic sound approximation to prevent divergence. There are more ways to convince Coq that a function does well-founded recursion, but they will not be covered in this tutorial.

Let us now define the addition of two natural numbers: `add`.

```
Fixpoint add n m :=
  match n with
  | O => m
  | S n' => add n' (S m)
  end.
```

The `add` function recurses over `n`. Since the recursive call is applied to the predecessor of `n` in the recursive case, Coq's type system figures this out, so we do not have to annotate explicitly on which argument the function is decreasing. The `add` definition performs a case analysis on its first argument. If it is zero, as there is nothing more to add, the function returns the second argument. If it is the successor of a number, we perform the addition of the predecessor of the first argument, and the successor of the second argument. Thus, `add` subtracts one unit to the first argument and adds one unit to the second argument, until the first argument is zero.

Coq makes available the keywords `Theorem`, `Lemma`, `Remark` and `Corollary` to state propositions. It also has the `forall` construction, which lets users generalize over an object. To prove properties Coq provides different tactic languages, in this document we use *ssreflect*. As an example property, we can state and prove that zero is the neutral additive on the left.

```
Theorem add0_n : forall (n : nat), add O n = n.
Proof. by []. Qed.
```

With the statement of the first line on the program above, we are instructing Coq to declare a `Theorem` under the name `add0_n`, holding a proof that `forall n : nat, add O n = n`. Then Coq opens a goal with that type, and we use the `Proof` keyword to start the proof, and enter in *proof mode*. In this mode the user may apply tactics and modify the state of the goal until there is nothing else to prove. This proof only requires unfolding the definition of `add`, which is done using the tactic `by`. The `by []` tactic unfolds definitions and computes in the goal until reaching the state of `n = n`. When the tactic is executed Coq computes on the left side of the equation. During the computation `add` matches on the first argument, it finds `O` and returns the second argument. This leaves the goal as `n = n`, which is true by reflexivity.

In Coq's setting, one says that "`p` is a proof of `P`" when `p` is a term of type `P`. In this context, `add0_n` is a proof stating that `O` is the left neutral additive for the natural numbers. The Coq tactic `Check` allows the user to query `add0_n`'s type.

```
Check add0_n.
(* add0_n : forall n : nat, add 0 n = n *)
```

In Coq every term has a type, and every type has a type as well. Above we see the type of `add0_n`, but what is the type of `forall n : nat, add 0 n = n`? If we query Coq for it the answer will be that it has type `Prop`, the type of propositions.

Coq provides different types of reasoning, for example the user is allowed to reason by induction. For any property `P` on the natural numbers it suffices to show two proofs. First, that the property holds for zero (i.e. `P 0`); and second, that for every natural number `n` satisfying the property (`P n`), its successor also satisfies it `P (S n)` (i.e. `forall n, P n -> P (S n)`). We present just below an example of a proof by induction.

```
Theorem addn_Sm : forall n m, add n (S m) = S (add n m).
Proof.
  move=> n.
  elim: n.
    + by [].
    + by move=> n' IHn m; rewrite -IHn.
Qed.
```

The theorem above is stating that for every two numbers `n` and `m`, adding `n` with the successor of `m` is equal to the successor of the addition of the two numbers. From now on we will display the proof state with the structure presented in Listing 1.

```
X goal
c_i : T_i
...
==================
forall (t : T),
P0 -> ... ->
C
```

Listing 1: Structure of a proof state.

In Listing 1, `X` stands for the number of goals. The section above the line is going to be called the *context*. In the context, `c_i : T_i` is a constant named `c_i` with type `T_i`. The section below the line is going to be called the *goal*. Finally, the `C` located at the end of the goal stands for the *conclusion*: what needs to be proved.

When opening the proof mode we have the following state of the proof:

```
1 goal
==================
forall n m : nat, add n (S m) = S (add n m)
```

Then, with the `move` tactic, we introduce the top term of the goal to the context.

```
1 goal
n nat
==================
forall m : nat, add n (S m) = S (add n m)
```

Now we have a particular `n` in context, and the goal generalizes over a natural number `m`. Having `n` in the context allows us to reason about `n` inductively with the `elim` tactic. After applying that tactic, Coq creates two subgoals ruled by the induction principle. The subgoals are presented below.

```
2 goals
==================
forall m : nat, add 0 (S m) = S (add 0 m)

subgoal 2 is:
forall n : nat,
  (forall m : nat, add n (S m) = S (add n m)) →
    forall m : nat, add (S n) (S m) = S (add (S n) m)
```

When there is more than one goal, bullet symbols (like +) indicate we are solving one of the subgoals. In the proof we are indicating that the first subgoal is solved with `by []`, and the second subgoal is solved with `by move=> n' IHn m; rewrite -IHn`. The first goal proof indicates that it can be solved by computation as we saw in our first example. The second subgoal is solved by introducing to the context `n' : nat`; `IHn`, referring to the induction hypothesis of type `forall m : nat, add n (S m) = S (add n m)`; and `m : nat`. This sequence of tactics leave the proof state as presented below.

```
1 goal
n' : nat
IHn forall m : nat, add n' (S m) = S (add n' m)
m : nat
==================
add (S n') (S m) = S (add (S n') m)
```

We figure that the right-hand side of the inductive hypothesis `IHn` appears on the right-hand side of the goal (after computation). With the `rewrite` tactic we can transform the goal rewriting under the equation `IHn`. This rewriting leaves the goal in a state that Coq can solve by computation: `add (S n') (S m) = (add n' (S (S m)))`.

Finally, we can prove zero is the additive neutral on the right, presented formally just below.

```
Theorem addn_0 : forall n, add n 0 = n.
Proof.
move=> n. elim: n.
 + by [].
 + by move=> n' IHn /=; rewrite addn_Sm IHn.
Qed.
```

The base case of the proof follows by computation. The proof for the successor is satisfied by a rewriting under our theorem `addn_Sm` and the inductive hypothesis. The `/=` symbol instructs Coq to compute after introducing `n'` and `IHn`.

2.4.3 Small Scale Reflection

We have been using ssreflect during the course of this introduction. SSReflect is a set of tactics developed to support the Small Scale Reflection methodology [26], which is a formal proof methodology based on the use of computation with symbolic representations. This methodology goes back and forth between the user and the prover. The prover provides computation, and the user guides step by step the execution.

The Calculus of Inductive Constructions distinguishes between logical propositions living in `Prop`, and boolean values living in the `bool` datatype. `Prop` supports natural deduction and `bool` allows case analysis. For cases in which the law of the excluded middle applies, there is a proof technique that allows the user to switch interchangeably between the two proof modes. Thus, the user can take advantage of both tools when appropriate, case analysis and reduction on the one hand, and natural deduction on the other. We show how to do this, first by defining a decidable predicate, such as equality between natural numbers, and then establishing a connection between that predicate and propositional equality.

We define the predicate `eqb_nat` to decide if two numbers are equal.

```
Fixpoint eqb_nat n m :=
  match n,m with
  | 0, 0 => true
  | S n', S m' => eqb_nat n' m'
  | _,_ => false
  end.
```

The `eqb_nat` fixpoint performs an equality test for two input numbers `n` and `m`. If both numbers match with `0` they are equal. If both numbers match with the successor of a number, `eqb_nat` compares the numbers recursively. In the two remaining cases (`0,S m'` and `S n', 0`) the numbers are different, this fact is indicated by the wildcard `_` placed on the last case, which matches with any pattern.

To establish the relation between `_ = _` and `eqb_nat _ _` we must use a special inductive named `reflect`, which we define below.

```
Inductive reflect (P : Prop) (b : bool) : Prop :=
| ReflectT (p : P) (bT : b = true)
| ReflectF (np : ~ P) (bF : b = false).
```

The `reflect` type receives a proposition `P`, and `b` a `bool`; with them, it establishes two connections: first that when `P` holds, the `b` is `true`, and vice versa; and second, that when `P` does not hold (i.e., `~ P`), then the boolean `b` is `false`, and vice versa. The proofs formalizing the relation between a proposition `P` and a predicate `p` through `reflect`, usually receive the name of views. The type `reflect` has three lemmas which we will use to link the propositional equality of natural numbers with `eqb_nat`. The lemmas are: `Bool.ReflectT`, stating that: for establishing the connection between any proposition `P` and the boolean value `true`, it suffices to show that `P` holds; `Bool.ReflectF`, stating that: for connecting any proposition `P` with the boolean value `false`, it suffices to show that the negation of `P` holds; and `equivP`, which establishes the connection of any two equivalent propositions yielding the same boolean.

```
Check Bool.ReflectT : forall (P : Prop), P -> reflect P true.
Check Bool.ReflectF : forall (P : Prop), ~ P -> reflect P false.
Check EquivP : forall (P Q : Prop) (b : bool),
                reflect P b -> P <-> Q -> reflect Q b.
```

With these lemmas we prove that for every natural number `n` and `m`, the propositional equality between `n` and `m` reflects to `eqb_nat n m`, that is `reflect (n = m) (eqb_nat n m)`. The proof goes by induction on `n` the first argument, and then by case analysis on `m`, the second argument. This opens four goals, corresponding to the four inductive cases. In the first case, when both numbers are zero, the goal reduces to `reflect (0 = 0) true`. The conclusion matches the conclusion of `Bool.ReflectT`, when we instantiate the proposition `P` with `0 = 0`. The tactic `apply` compares the lemma with the current goal and as it matches, Coq replaces the goal with the hypothesis of the lemma. In this particular case, it means that we now have to show that `0 = 0` holds, which is solved automatically prepending the tactic `by`. The second and third case, corresponds to when one of the numbers is zero and the other one is the successor of a number. The predicate is `false` by the definition of `eqb_nat`, and the propositional equality does not hold as well, because `nat` constructors are disjoint. Hence, we apply the lemma `Bool.ReflectF`, whose hypothesis also happens to be automatically provable in these cases. In the last case, Coq reduces `eqb_nat (S n') (S m')` to `eqb_nat n' m'`, which can be solved by applying the inductive hypothesis and the use of `equivP`.

```
Lemma nat_eqb_eq : forall (n m : nat), reflect (n = m) (eqb_nat n m).
  Proof. move=> n. elim: n=> [| n' IHn] [| m'].
   + by apply Bool.ReflectT.
   + by apply Bool.ReflectF.
   + by apply Bool.ReflectF.
   + by move=> /=; apply: (equivP (IHn m') _); split=> [->|[->]].
 Qed.
```

The lemma nat_eqb_eq allows us to interchange between the propositional equality of natural numbers, and the predicate that decides their equality, indistinctly. We have already seen that for every natural number n, the proposition n = n can be solved automatically. This does not happen with our predicate, if we try to prove that eqb_nat is a reflexive predicate (i.e., eqb_nat n n always reduces to true) automatically, we would not be able to. Of the natural number n we only know its type, we do not know if it is zero or a successor. Therefore, Coq cannot determine which case of match should apply. If we wanted to prove eqb_nat n n then, we would have to reason inductively about n, but since we have the connection through reflect, we just need to exchange the predicate eqb_nat for the proposition of equality. Now we show how to use that in a proof, by proving that eqb_nat is a reflexive predicate by using the fact that n = n holds trivially. Now that we have proven the lemma nat_eqb_eq we can exchange the value of eqb_nat n n for its propositional equivalent n = n. To do so, we use the lemma introT, which indicates that for every proposition P and boolean value b, we can conclude that b is true by providing a proof of P and that P and b are related through reflect. This is precisely our case, we want to prove that eqb_nat n n is true, and we have already proven the connection through reflect. If we apply the lemma introT with the proof nat_eqb_eq as an argument, it suffices to show that n = n, which can be solved automatically.

```
Check introT : forall (P : Prop) (b : bool), reflect P b -> P -> b.

Lemma eqb_nat_refl (n : nat) : eqb_nat n n.
Proof. by apply (introT (nat_eqb_eq n n)). Qed.
```

This proof technique is used very frequently, so mechanisms have been developed to use it in a less verbose way. These mechanisms work through canonical structures [50], structures that allow type inference to be able to interchange propositions and predicates related through reflect.

2.4.4 Canonical structures

The Mathematical Components library makes available different interfaces with useful lemmas and notations about the objects. Making an instance of these interfaces transfers all notations and lemmas to the target type. This is enabled through a type inference mechanism called Canonical Structures [50]. For example, there is the eqType interface for objects with decidable equality. This interface gives the user access to notations like _ == _ notation; lemmas about equality, like the lemma contraNneq b x y: (x = y -> b) -> ~~ b -> x != y and views like eqP, which allows the user to go from x = y to x == y and vice versa. If one would like to make a type, let say nat, an instance of eqType one has to provide two things. First, an equality operator eq_op : nat -> nat -> bool, and second, a proof stating that eq_op corresponds with the propositional equality (i.e., that the propositional equality, and eq_op are in relation through reflect). We have already declared both requirements; they correspond to the predicate eqb_nat and the lemma nat_eqb_eq, which were defined and proved in the previous subsection. With this, we can declare natural numbers to be an instance of eqType.

```
Definition nat_eqType := EqType nat (EqMixin nat_eqb_eq).
Canonical nat_eqType.

Variables n m : nat.
Check n == m.
(* n == m : bool *)
```

The canonical definition we made declares nat to be an instance of eqType. By defining this instance we are instructing Coq to implicitly go through the eqType interface whenever it sees a proposition holding for eqTypes applied to a natural number. Through this definition we make all lemmas and notations available for eqType to also be available for natural numbers. As an example, we declare two natural numbers n and m, and compare them using eqType's notation for equality (_ == _). It is worth mentioning that Coq's type system notes that nat_eqb_eq has type forall x y, reflect (x = y) (eqb_nat x y), and hence, nat_eqType is completely specified without having to give eqb_nat as an argument. We now provide an example using the math-comp interface for eqTypes. We prove the transitivity of the eqb_nat predicate.

```
Lemma eqb_nat_trans (n m l : nat) : eqb_nat n m -> eqb_nat m l -> eqb_nat n l.
Proof.
move/eqP=> eqnm.
by rewrite eqnm.
Qed.
```

The tactic move under the view /eqP reflects the boolean equality eqb_nat into the propositional equality =. Leaving the goal like this: n = m -> eqb_nat m l -> eqb_nat n l. After that => let us introduce a hypothesis from the goal to the context. We introduce the equation n = m with the name eqnm, and right after we rewrite the goal under that equation. This leaves the goal as follows: eqb_nat m l -> eqb_nat m l. This is trivially solved by Coq as the conclusion is within the hypothesis.

The pattern of applying a view, obtaining an equation, and immediately using it to rewrite is a common pattern in the methodology. If the hypothesis at the top of the goal is an equation, instead of naming the hypothesis the user may use the -> symbol. This instructs Coq to rewrite the goal under the top equation of the goal. Using that pattern would leave the proof script as: `by move/eqP ->`, which is a more succinct and elegant solution.

2.4.5 Records and hierarchies

Records are a data structure. They are composed of named fields. In the dependently typed setting one may let fields depend on the value of other fields. Let us make it clear with an example; consider a primality test `is_prime : nat -> bool`. We may define the types of natural numbers, which are primes.

```
Record prime : Type := mkPrime {
  p :> nat;
  p_prime : is_prime p
}.
```

The record `prime` is a `Type` with only one constructor `mkPrime`, which requires a natural number `p`, and a proof stating that `p` is prime (`is_prime p`). When a specific term requires a proof to be built, we say that term is a proof carrier, for example, since to build a `prime` we need a proof of `is_prime p`, we say `prime`s are proof carriers. Considering that `is_prime p` is a `bool`, this should not type-check. But, Coq figures this out and transports the value into `Prop` by silently inserting the function `is_true: bool -> Prop`. The symbol :> after `p` instructs Coq to silently project `p` if Coq expects a natural number, but receives a `prime`. A function that Coq silently and automatically inserts in order to transport terms over types receives the name of *coercion*.

Part of the ssreflect methodology is to take advantage of the canonical structures to define hierarchies. This way, particular instances of the objects also inherit the benefits of the interface. As a term of type `prime` can be coerced to the natural numbers, we would like `prime`s to also enjoy the benefits of `eqType` through its connections with natural numbers. To do that we will define a way to *encode* and *decode* natural numbers into primes, and also primes into natural numbers in a way that the decoding cancels the encoding. For doing this, we must show that a prime `p` is equal to `decode (code p)`. The equality of records is defined as the equality of each of their fields. In the case of primes, this is the equality of the underlying natural number, and the equality of the proof stating that the number is prime. But indeed the equality of the underlying numbers suffices to show that the primes are equal, we prove it below. We will use this lemma to prove that the encoding and decoding cancels out.

```
Lemma prime_inj (p1 p2 : prime) :
  p p1 = p p2 ->
  p1 = p2.
Proof.
  case p1; case p2=> prime1 prime1P prime2 prime2P /= eq.
  subst; congr mkPrime.
  by apply eq_irrelevance.
Qed.
```

The proof starts by reasoning by case analysis over the two primes `p1` and `p2` and introducing the resulting subterms to the context: two natural numbers `prime1` and `prime2`, a proof that `prime1` is prime `prime1P`, a proof that `prime2` is prime `prime2P`, and the equation `eq : prime1 = prime2`. We then perform a substitution using the equation `eq`. We cannot perform a rewrite because the conclusion would be thus ill-typed. The interested reader can look up for rewriting in dependent contexts. It is then left to prove that the two proofs of `is_prime prime1` are equal, i.e. to prove that the equality `prime1P = prime2P` holds. This is always the case when the proofs are in `bool`. The equality comes from the coercion `is_true`, which after computation ends with `is_prime prime1 = true`. This interesting property stating that: for any two proofs of an equality, the two proofs are equal; holds for any `eqType`. This is why equality on `eqType` is said to be *proof irrelevant*. That property is formalized by the Mathematical Components library with the lemma `eq_irrelevance`, whose type is displayed below.

```
Check eq_irrelevance.
(* eq_irrelevance : forall (T : eqType) (x y : T) (e1 e2 : x = y), e1 = e2 *)
```

The type of `eq_irrelevance` states that: for any two terms `x` and `y` of type `T : eqType`, for any two proofs stating `x` and `y` are equal, the proofs are equal. Now that we have proven the injection lemma, we can provide the `code` and `decode` functions to map primes to natural numbers, and vice versa.

```
Definition code (p : prime) : nat :=
  p.

Definition decode (n : nat) : option prime :=
  if insub n : {? x | is_prime x} is Some nn
  then Some (mkPrime (valP nn))
  else None.
```

To code the `prime` type to the `nat` type, there is nothing to be done. Remember that when defining the record we instructed Coq to insert a coercion to `nat`. From the type of the `code` function Coq expects the output to be a `nat`, but a prime is returned instead. Coq realizes this and projects the natural number from the record.

When decoding a natural number there is a (high) chance it does not satisfy the primality test. If the `decode` function was total we need to return a `nat` even when it is not the case. But, if we choose a default result there will be no guarantee that the decoded prime corresponds to the natural. We can implement such a behavior with a partial function. Partial functions in Coq are modeled using the option type.

```
Inductive option (A : Type) := None | Some : A -> option A.
```

Option is a type constructor, meaning that it builds new types from previously defined ones. It is also polymorphic, i.e. `option` is parameterized by a type `A`, which can be instantiated with *any* type. The instantiated type enjoys all the lemmas and operations defined for `option`. Given a type `A`, the type `option A` has two constructors, `None` and `Some`. We can use the `option` type to model partial functions in Coq. In particular, we can use it to model our `decode` function with the type `nat -> option prime`. This is the idea: if `decode` receives a natural number, which satisfies the primality test; we can return a prime wrapped with the `Some` constructor, and if the input does not satisfy the primality test we return `None`. Now we can prove `code` and `decode` cancel each other out, whenever the decoding returns a `Some` value.

```
Lemma pcancel_code_decode : pcancel code decode.
Proof.
  case=> p pP /=; rewrite /decode.
  case: insubP => [? ? ?|].
  by congr Some; apply prime_inj.
  by rewrite pP.
Qed.
```

The proof goes by case analysis on the result of the primality test. If the test passes we can apply the injection lemma we proved before. If it does not pass, we have a contradiction.

Finally, we can derive the canonical instance for primes.

```
Definition prime_canEqMixin := PcanEqMixin pcancel_code_decode.

Canonical prime_eqType :=
  Eval hnf in EqType prime prime_canEqMixin.

Variables p1 p2 : prime.
Check p1 == p2.
```

When the type we want to code does not have an obvious target type there is still a solution. The Mathematical Components library defines the `GenTree` inductive. `GenTree` inherits several interfaces, therefore, if a programmer manages to encode and decode their type into the `GenTree` type, they can follow the same methodology to derive the canonical instance.

2.4.6 Math-comp notation

Coq enables the definition of notations. With these notations, a user can extend Coq's syntax for an alternative way of entering or displaying a term. During the development of this work we will use some notations defined by Coq and math-comp; here we indicate the meaning of these notations. Consider that `P1` is convertible to `forall x, Qx`, and that `P2` is convertible to `forall x y, Qxy`. Then, we use the following notations.

- `[seq x <- s | C] := filter (fun x => C) s.`
- `[seq E | x <- s] := map (fun x => E) s.`
- `(x \in P) := P x.`
- `{in A, P1} := forall x, x \in A -> Qx.`
- `{in A &, P2} := forall x y, x \in A -> y \in A -> Qxy.`
- `A =i B := := forall x, x \in A = x \in B.`
- `f1 =1 f2 := forall x, f1 x = f2 x.`

2.5 Summary

In this chapter we have introduced the notions of ITPs. We have chosen to work with the Coq proof assistant using the Mathematical Components libraries, and hence adopting the ssreflect methodology. Examples of mechanically formalized projects were presented. The chapter presented a tutorial exposing some features of the Coq proof assistant by example. We also exemplified the use of the ssreflect methodology using the Mathematical Components interface infrastructure. The tutorial goes from the definition of inductives and functions to the proof of theorems, and different approaches to transferring canonical structures.

In the next part, RDF, the object of study, is going to be presented in detail, followed in the subsequent chapter by a mechanized implementation of its components.

Part II

Formalizing the RDF model

This part covers the presentation of the Resource Description Framework (RDF) and our mechanized implementation of the RDF model using the Coq proof assistant. In Chapter 3 we introduce RDF's background. An historical context and motivation of RDF is described in Section 3.1. We then point out how the RDF model has changed during the last few decades in Section 3.2. Subsequently, we present RDF's definition and abstract representation in Section 3.3 with a gradual introduction of RDF's core components: RDF terms, triples and graphs. We explain how these components interact, and how they are used to describe resources. Afterwards we highlight the complexity of blank nodes, and RDF's operations using them in Section 3.4.

In chapter 4 we introduce `CoqRDF`, a library we have developed, which defines the RDF model. We implement `CoqRDF` in the proof assistant Coq using the ssreflect tactic language and methodology, and the Mathematical Components libraries of formalized mathematics. `CoqRDF` gives a mechanized implementation of the RDF model. We present a well-formedness specification of every RDF component, and then prove our definitions satisfies them. Within `CoqRDF` we define RDF's terms, triples and graphs as Coq mathematical objects in Sections 4.1, 4.2 and 4.3 respectively. These definitions allow users to define RDF operations on top of it, and to pose correctness specifications of their definitions. We define the notion of blank node mappings, which we specify in Coq, and then we provide an implementation meeting that specification.

Chapter 3

The Resource Description Framework

In this chapter we cover the Resource Description Framework (RDF). In Section 3.1 we present the context, problems and motivations encouraging the rise of RDF. Time makes things change, and we follow how RDF has evolved through its different versions in Section 3.2, we highlight some changes during the history of RDF's development, and how it was proposed as a W3C standard which contributes to the stability and accessibility of the World Wide Web. We then continue with an introduction to RDF's model in Section 3.3 where we address RDF's abstract syntax representation by explaining its structural composition. We present RDF terms, triples and graphs, and how to use them for describing resources. In Section 3.4 we problematize the definition of blank nodes. We present how blank nodes interact within the RDF model as per RDF graph isomorphism, and also with other standards relying on RDF as in RDF's test suites and serializations.

3.1 RDF evolution

The World Wide Web (WWW) has been growing since its creation in 1989. Although every document in the WWW is machine-readable, back then, they were not machine-understandable, meaning that the documents were difficult to process. The problem was that every document had to be processed manually, and because of that, it was unachievable to automatically process the amount of documents available on the WWW. To address this problem, Lassila and Swick [44] proposed RDF to describe the data contained on the Web using metadata, i.e., data about data.

The RDF model was initially proposed in 1997 [43] as a foundation for processing metadata. After two years of discussion, RDF was recommended by the World Wide Web Consortium (W3C) [44] in 1999, meaning that RDF was being considered to become a standard for processing metadata on the WWW. As per W3C's recommendation, RDF's goal is to provide interoperability between applications that exchange information on the web. Particularly, RDF's design allows us to understand a document's structure and logical content, which improves a search engine's opportunities to provide better results. It also allows us to join collections of web pages representing a single logical document, and to describe intellectual property rights of Web pages, etc.

RDF is a model intended to be used in situations where data is going to be processed, not just displayed. As it is critical when publishing and interlinking data on the Web, RDF provides exchange between applications without loss of meaning. Its name already tells us its function: RDF is a framework for describing resources, where resources can be anything such as documents, people, abstract concepts, etc.

3.2 RDF history

RDF was defined more than 25 years ago. Throughout the decades of its development RDF has extended its definition for two main reasons: to improve logical reasoning about the RDF model, and to improve its usage by taking into account new technologies. Let us review how RDF has dealt with these challenges. The original 1999 proposal provided RDF syntax, serialization, and its formal model and grammar [44]. The recommendation document was oriented to define how to specify data semantics using the Extensible Markup Language (XML). This design choice tied RDF to XML. RDF's goal was to design a domain neutral mechanism to model data, that is RDF would not make particular assumptions of its application domain, nor define the semantics of any application domain, but RDF should be able to describe any domain. Because of RDF's definition universality, its use was explained with examples in different domains.

RDF was born as a device to add machine-readable metadata to the Web, with the goal of providing a lingua franca for the automated processing of Web information. This was achieved in 2004, then RDF 1.0 [11] was formalized as a standard recommendation by the World Wide Web Consortium (W3C). The W3C standards are Open Web Platforms for application development with technical specifications. RDF took the opportunity to go one step further: it provided a basis for well-founded deductions about RDF data. In its version 1.0 RDF introduced its formal semantics, and abstract syntax to link the concrete syntax with the formal semantics. These additions provided support for two things: first, to formally reason about the meaning of an RDF expression, and to define inference rules about RDF data. For example, by adding URIs, RDF improved its clarity in the cases where the described resource lived in the real world, instead of living as a document on the Web, as how it was thought when RDF was originally designed. The use of URIs provides identifiers to reference resources in the real world, which helps users distinguish the resource of a website from the resource which a website makes reference to.

RDF's specification stood stable for a decade, until 2014 when it was updated to version 1.1 [18]. The new version was designed to be compatible with RDF 1.0 while generalizing over its own definitions. For example, RDF 1.1 introduced a new encoding to globally identify variables: the International Resource Identifier (IRI). This change, among others, improved RDF's support for internationalization by adding support for a broader range of Unicode characters. Regarding semantics, RDF 1.1 included an updated semantics description. This updated semantics defined new entailment regimes [18] to infer the meaning of RDF data. Additionally, RDF introduced user-friendly syntaxes and new serialization formats like Turtle, JSON-LD, N-Triples, TriG, etc.

Over the two decades of RDF design many trade-offs were made; and some of these decisions were never completely summarized in the RDF standards [71]. However, RDF keeps gaining popularity. Its adoption has been increasing: 38%-45% of websites are using RDF syntaxes such as JSON-LD or RDFa [69], and 64% of the websites are using structured data based on RDFa.

3.3 The RDF 1.1 model

The RDF 1.1 model, from now on just RDF, is a graph-based data model, using nodes and edges to describe resources. Let us gradually introduce RDF's components and present how they are used to describe a resource. In RDF, nodes and edge labels (known as *properties*) can be of three types: *IRIs* globally identifying an entity, such as a university, for example the University of Chile; *literals* of various datatypes such as numbers, dates or strings; and *blank nodes*, which represent existing variables with an unknown value. The set formed by the union of these types is the set of *RDF terms*, and they are used to express information by making *statements*. Statements are three-element tuples with the following structure:

$$< subject >< predicate >< object > \mid \begin{array}{l} \textit{The subject is an IRI or a blank node}, \\ \textit{the predicate is an IRI}, \\ \textit{the object is an IRI},\ \textit{a literal or a blank node}. \end{array}$$

Statements are collectively known as *RDF triples*, and are conventionally written in the order: subject, predicate, object. Given three RDF terms `a`, `b`, and, `c` we note a triple with `a` as subject, `b` as predicate, and `c` as object, as (`a`, `b`, `c`). RDF triples allow us to relate two resources. The *subject* and the *object* represent the two resources being related; and the *predicate* represents the nature of their relationship. In RDF, relationships are phrased from subject to object and the kind of relation is indicated by the IRI placed in the predicate position.

For example, if Alice is friend of Bob we can model that with the following RDF triple: (`:Alice`, `:isFriendOf`, `:Bob`)[1], given two IRIs `:Alice` and `:Bob` for identifying our individuals, and an IRI for identifying the friendship relation `:isFriendOf`. Within that triple `:Alice` would be the *subject*; `:isFriendOf` the *predicate* and `:Bob` the *object*. That triple can be drawn as a labeled graph as shown in Figure 3.1. In this graph abstraction, subject and object terms refer to nodes, predicate terms to edge labels, and a triple to a directed labeled edge. From now on, whenever we draw an RDF triple IRIs are going to be colored in purple and prefixed with ":", blank nodes in light blue and prefixed with "_:", and literals in yellow and without prefix.

[1]In order to assist readability some serialization formats allow prefixes definition. For example, defining the prefix : to be http://ex.org, then `:Alice` is a shortcut for http://ex.org/Alice. However, these prefixes are not a formal part of the RDF data model, just a syntactic convenience. We use the default prefix : in our examples where unimportant.

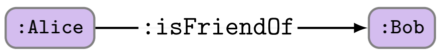


Figure 3.1: A statement describing Alice and Bob's friendship.

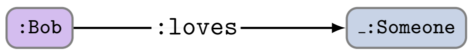


Figure 3.2: A statement claiming that Bob is in love with someone.

Similarly, if we know that Bob is in love with someone, but we do not know specifically with whom he is in love, we may state this relation using the IRI `:loves` as predicate and a blank node `_:Someone` as the object. The RDF triple (`:Bob`, `:loves`, `_:Someone`) uses a blank node to represent this fact for which we know the data only partially. When we have a set of RDF triples we refer to it as an *RDF graph*, and a collection of RDF graphs is called an RDF dataset. Let us have a look at an RDF graph in Figure 3.3.

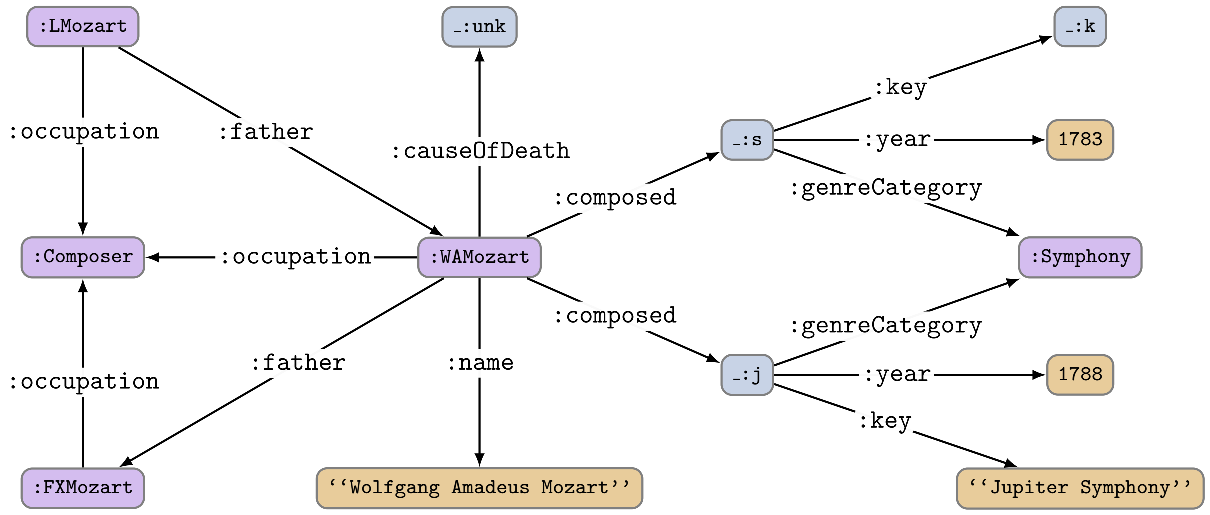


Figure 3.3: An RDF graph describing Wolfgang A. Mozart.

In Figure 3.3 `_:s`, `_:j`, `_:k`, and `_:unk` are blank nodes. The graph states a family of composers. `:LMozart` is the father of `:WAMozart`, who is father of `:FXMozart`. It also presents two works of `:WAMozart`, `_:s` and `_:j`, which are unidentified. But, we do know certain properties about these works. Both are symphonies, composed in different years. We also know `_:s` was composed before `_:j`, and that `_:j`'s nickname is `''Jupiter Symphony''`.

3.4 RDF isomorphism

Blank nodes give rise to a binary operation to test for structural equality between two graphs, which the RDF 1.1 recommendation [18] defines as *RDF graph isomorphism*.

Definition 3.1 (RDF graph isomorphism) *Two RDF graphs G and G' are isomorphic (that is, they have an identical form) if and only if there exists a bijection M between the sets of nodes of the two graphs, such that:*

- *M maps blank nodes to blank nodes.*
- *$M(lit) = lit$ for all RDF literals lit which are nodes of G.*
- *$M(iri) = iri$ for all IRIs iri which are nodes of G.*
- *The triple $(s,\ p,\ o)$ is in G if and only if the triple $(M(s),\ p,\ M(o))$ is in G'.*

With this definition, M shows how each blank node in G can be replaced with a new blank node to give G'.

Testing isomorphism between two graphs is not an easily solved problem, but let us illustrate with an easy example in Figure 3.4.

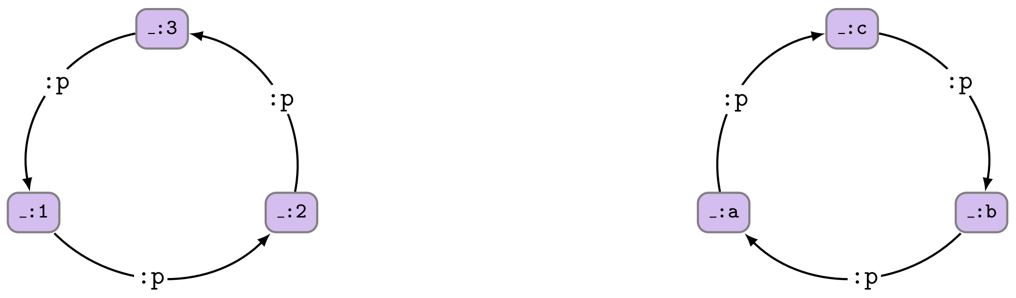


Figure 3.4: Two isomorphic RDF graphs. The bijection M is $M(1) := a$, $M(2) := b$, and $M(3) := c$.

As we can see, besides the fact that the two graphs on Figure 3.4 are drawn differently and have different blank nodes, the two graphs are structurally identical. The presence of blank nodes complicates things. Let us recapitulate its definition, the RDF standard defines blank nodes as existential variables, but RDF's specification makes no reference to any internal structure of blank nodes; besides that it states that blank nodes do not have identifiers in the RDF abstract syntax, but that they are distinguishable. Furthermore, the specification about blank nodes has been pointed out to be unclear since RDF's version 1.0 [52], and then during the discussion of the W3C working group in charge of defining RDF's version 1.1 [36]. In the analysis of blank nodes made by Mallea et al. [52], and its extended version by Hogan et al. [36], they found that blank nodes are treated differently (and sometimes incompatibly) across other standards built on top, or related to RDF. For example, when serializing an RDF graph, any of RDF's syntaxes allows to explicitly label blank nodes, such that they can be referred to at any point within the document [36]. These syntaxes include RDF/XML [6], N-Triples [63], turtle [10], RDFa [30] and JSON-LD [66]. This is an issue, because syntaxes allow syntactic sugar, which for some constructions introduce implicit blank nodes (as it is the case when using RDF lists), and RDF parsers can arbitrarily assign blank node identifiers. This condition forces that tests suits have to check their results modulo isomorphism [42], but the declarative style of the RDF's isomorphism definition does not leave an obvious implementation algorithm to test it.

3.5 Summary

In this chapter, we presented RDF, a graph data model for enriching the documents of the WWW with machine-readable data. We give historical context to understand the state of the Web, and people's issues and goals when processing web documents at the time of RDF's proposal. We described RDF's different versions, where it was originally recommended by the W3C in 1999, a standard that was replaced with the release of RDF 1.0 in 2004, and later upgraded with the recommendation of RDF 1.1 in 2014, its current version. We pointed out the main reasons for updating the RDF standard and how these reasons have been handled by RDF's different versions.

We then introduce RDF's abstract syntax and its core components: RDF terms, triples, and graphs. Later, we explain how to use these components to describe resources, which leads us to some problems regarding the blank node's definition. The presence of blank nodes gives birth to the notion of RDF graph isomorphism, a key operation to define RDF's behavior, but with an unclear operational definition.

In the next chapter we present `CoqRDF`: the first mechanized implementation of the RDF data model. We define `CoqRDF` within the Coq proof assistant using the Mathematical Components library. We declare RDF well-formedness specification, which our components satisfy by construction. Our implementation allows us to reason about RDF terms, triples and graphs. These definitions are then used to define mappings between blank nodes. We study the behavior of these mappings and formalize identities around the structure of RDF graphs in the presence of blank nodes. This effort is directed towards an operational definition of isomorphism, which is addressed in part III.

Chapter 4

`CoqRDF` : a mechanized definition of RDF

In chapter 3 we described the RDF model and its main components: terms, which are its core bricks; triples, which are terms triples, with specific restrictions about their possible combinations; and RDF graphs, which are sets of triples. We also presented the notion of RDF isomorphism, a binary relation denoting structural equality between graphs.

In this chapter we cover mechanical definitions of every component of the RDF model, as well as a few abstractions surrounding these components. We incrementally tackle the modeling of RDF's abstract representation. In parallel, we define the notion of blank node relabeling, and explore how this notion interacts with our definitions. In each section in this chapter we implement an RDF component, define how to relabel blank nodes within that scope, and end with an example using `CoqRDF`. We mainly study the action of blank node replacement under different assumptions, but we also explore identities of other abstractions less important to this work. RDF terms, triples, and graphs are addressed in Sections 4.1, 4.2, and 4.3 respectively. We implement these mechanical definitions using Coq and the library of Mathematical Components. We implement an encoding for each RDF component in order to transfer different math-comp's theories to our definitions. Our development, called `CoqRDF`, is available in a GitHub repository at *https://github.com/Tvallejos/CoqRDF*.

4.1 `CoqRDF` Terms

In this section we define RDF terms. We declare an inductive type characterizing the different kinds of nodes in Subsection 4.1.1. We then study the requirements for equipping this inductive type with: equality, countability, choice and order operations in Subsection 4.1.2. Afterwards we make remarks about what can (or cannot) be expected from terms' equality in Subsection 4.1.3. Finally, we model the relabeling of blank nodes at the scope of terms in Subsection 4.1.4, present an illustrative example in `CoqRDF`, and study its properties under different assumptions of terms' structure in Subsection 4.1.5.

4.1.1 Terms definition

We begin by defining the core of the RDF model: terms. As discussed in Section 3.3, terms can be of three kinds: *Iris*, *Literals* and *Blank nodes*. To reflect this in our representation we model terms as an inductive type whose definition is presented in Listing 2.

```
Inductive term (I B L: Type) :=
| Iri (i : I)
| Bnode (b : B)
| Lit (l : L).
```

Listing 2: RDF terms defined as an inductive type.

Term's inductive has three constructors, one for every kind of term. This inductive is parameterized by three types `I`, `B`, and `L`, which represent the set of IRIs, blank nodes and literals, respectively. These parameters make every terms' constructor parameterized by those same parameters. Every constructor requests an inhabitant of the respective parameter as an argument in order to return it as a term. A direct consequence of our characterization is that every constructor returns a uniform type, i.e. every term's constructor returns a term instantiated with its type parameters. Let us illustrate. When a `term`'s constructor is instantiated with the types `I B L`, and an inhabitant of one of these types, the constructor produces a `term I B L`. Thus, the inductive `term` brings us a uniform interface to deal with terms: `term I B L`. This interface, has as drawback that we cannot define a literal (or any other kind of term) if we have not specified every type parameter, i.e. for defining any kind of term, we need to know which are types of the IRIs' and blank nodes' and literals, as can be seen in Listing 3 where we query every constructor's type.

```
Check @Iri.
(* Iri   : forall I B L : Type, I -> term I B L *)
Check @Bnode.
(* Bnode : forall I B L : Type, B -> term I B L *)
Check @Lit.
(* Lit   : forall I B L : Type, L -> term I B L *)
```

Listing 3: Queries to every term's constructor type.

4.1.2 Terms encoding

We now transfer some of math-comp's theories to our `term` inductive. To achieve that we start by defining an encoding from `term I B L` to `GenTree.tree`. The `GenTree.tree` defines once and for all how math-comp's theories behave, and by defining an encoding to this type we can transfer the different theories to the type `term`. For example, assuming that the set of IRIs, blank nodes, and literals can be compared with equality, can be counted, have a canonical choice and have a partial order, we may extend these operators to the `term`'s inductive through the encoding to `GenTree.tree`. As a result, we can define terms as an instance of eqType, countType, choiceType and POrderType. For all these mentioned instances all we need is the encoding, and that the term's type parameters are already equipped with the instance we are trying to declare. We choose to define the encoding of terms as a sum of the three type parameters: I + B + L, which gives us a direct way to decode encoded terms back to a term I B L. First, when encoding a term, whichever the kind of node it is, let us say it is an Iri i: we create a GenTree.Leaf with the only possible injection we can build from an i : I to I + B + L, i.e. inl (inl i). Then, when decoding a GenTree back to a term; we can recover the type parameters of the term we want to return from the GenTree's type, mapping back every injection to its original constructor. In the case of IRIs we map GenTree.Leaf (inl (inl i)) : GenTree.tree (I + B + L) back into an @Iri I B L i. Finally, we prove a lemma that decoding after encoding a term cancels the encoding out; since not every GenTree can be decoded into a term (because GenTree has more constructors) the cancellation is only partial. The complete encoding is presented in Listing 4.

```
Section CodeTerm.
  Variables (I B L : Type).
  Implicit Type trm : term I B L.

  Definition code_term trm : GenTree.tree (I + B + L)%type :=
    match trm with
    | Iri i => GenTree.Leaf (inl (inl i))
    | Lit l => GenTree.Leaf (inr l)
    | Bnode name => GenTree.Leaf (inl (inr name))
    end.

  Definition decode_term (x : GenTree.tree (I + B + L)) : option (term I B L) :=
    match x with
    | GenTree.Leaf (inl (inl i)) => Some (Iri i)
    | GenTree.Leaf (inr l) =>  Some (Lit l)
    | GenTree.Leaf (inl (inr name)) => Some (Bnode name)
    | _ => None
    end.

  Lemma pcancel_code_decode : pcancel code_term decode_term.
  Proof. by case => [i | l | name]. Qed.

End CodeTerm.
```

Listing 4: Term's encoding to GenTree.

4.1.3 Terms equality

We now cover what we can expect from the term's equality operator. Because we are immersed in a strongly typed environment, Coq's type checker prevents us, statically, from

```
Fail (@Lit bool nat nat 3) == (@Lit nat nat nat 3).
(* indeed fails with type error ... *)

Compute (@Iri nat nat nat 1) == (@Lit nat nat nat 1).
(* = false : bool *)

Compute (@Iri nat nat nat 5) == (@Bnode nat nat nat 5).
(* = false : bool *)

Compute (@Lit nat nat nat 0) == (@Lit nat nat nat 0).
(* = true : bool *)
```

Listing 5: Examples using term's equality.

performing certain comparisons. For example, it does not type check to compare two `terms`, which are parameterized with different types. We present this in Listing 5, where the first line fails because Coq expects a term of type `term bool nat nat` on the right side of the equality (==). This happens even though we are comparing a literal representing the same number: `3`. When comparing `terms` parameterized with the same types, equality behaves as specified in RDF's recommendation [18]. We have, for example, that the literal `1` is different from the IRI `1`, and that the IRI `5` is different from the blank node `5` whereas the literal `0` is equal to itself. All these examples are presented as well in Listing 5.

However, because of the encoding we defined for declaring the `term`'s `eqType` instance, Coq has to go through that encoding to compute equality. This may add some noise when proving properties involving `term`'s equality. To circumvent this issue we define an equivalent function: `eqb_term`, which performs equality without going through the encoding. We define it as follows: two terms of different kinds are different, and if both terms are of the same kind, and they represent the same literal, IRI or blank node, those terms are equal. We prove under the lemma `eqb_eq` that for every two terms the equality under == and `eqb_term` are equivalent. When proving the identity of the equivalences, the proof from left to right is followed by case analysis after rewriting under `trm1 = trm2`. From right to left after case analysis for both terms, and after computation, we have the conclusion as an assumption. We present the lemma `eqb_eq` and symbolic equality function `eqb_term` in Listing 6.

```
Section EqTerm.
  Variables I B L : eqType.

  Definition eqb_term (t1 t2: term I B L) : bool :=
    match t1, t2 with
    | Lit l1,Lit l2 => l1 == l2
    | Bnode b1, Bnode b2=> b1 == b2
    | Iri i1, Iri i2 => i1 == i2
    | _,_ => false
    end.

  Lemma eqb_eq trm1 trm2 : (trm1 == trm2) = eqb_term trm1 trm2.
  Proof.
    apply /idP/idP.
      + by move=> /eqP ->; case: trm2=> ? /=; rewrite eqxx.
      + by case: trm1; case:trm2 => //= t1 t2 /eqP ->; rewrite eqxx.
  Qed.

End EqTerm.
```

Listing 6: Definition of term's symbolic equality (`eqb_term`) and a proof of its equivalence with ==.

4.1.4 Terms relabeling

We have equipped `terms` with equality, countability, choice and order operations in Subsection 4.1.2. Now, towards defining RDF isomorphism we define *blank node mappings*, capturing the notion of *blank node relabeling*, the action of replacing a blank node with another one.

Definition 4.1 (Blank node mapping) *Given a function μ from RDF terms to RDF terms, we say it is a blank node mapping if and only if:*

- *μ maps blank nodes to blank nodes.*
- *$\mu(lit) = lit$ for any literal lit.*
- *$\mu(iri) = iri$ for any IRI iri.*

Definition 4.1 captures the first three clauses of RDF isomorphism (3.1). However, a blank node mapping is a relaxed version of an RDF isomorphism: the function μ does not have to be bijective nor preserve membership when applying μ to a graph, as is the case for RDF isomorphism. Note that every RDF isomorphism is a blank node mapping. In Listing 7 we define a blank node mapping builder: `relabeling_term`, a function lifting a map between two types of blank nodes to a map between `terms` parameterized with different types of blank nodes. We define relabeling in such a way that for every function `mu : B1 -> B2` then relabeling a `term` under `mu` meets the conditions of a blank node mapping. The function `relabeling_term` receives four types: the type of IRIs (`I`) and literals (`L`), and the type of the domain (`B1`) and codomain (`B2`) of the relabeling. Then `relabeling_term` asks for a function `mu` going from the domain to the codomain, and returns a function which takes a term `trm` of type `term I B1 L` and returns a `term I B2 L`. If `trm` is an IRI or a literal, then `relabeling_term mu` returns the same value, and if `trm` is a blank node it applies `mu` and wraps the result in a `Bnode`.

```
Definition relabeling_term (I B1 B2 L : Type) (mu : B1 -> B2)
  (trm : term I B1 L) : term I B2 L :=
  match trm with
  | Bnode b => Bnode (mu b)
  | Iri i => Iri i
  | Lit l => Lit l
end.
```

Listing 7: A function lifting mappings between two types to mappings between two types of terms.

We now verify that for every function `mu`, then it holds that `relabeling_term mu` is a blank node mapping. Nonetheless, the definition of `relabeling_term` is too general for the property we want to verify. As the type of the domain and codomain of `mu` can be different, the type of the resulting term can be different as well, and as discussed in Listing 5 we cannot compare with equality two terms that are parameterized by different types. However, we can restrict `B1` and `B2` to be the same type `B` and prove that for every `mu : B -> B`, it follows that `relabeling mu` satisfies the blank node mapping specification.

We formalize in Coq the blank node mapping's definition (4.1). We indicate the abstractions we need to state the proposition of every item in the definition. The first clause requires a predicate `is_bnode`, which decides if a term is a blank node. With this predicate we have to prove that `is_bnode` returns `true` whenever we relabel a term which was a blank node. In the case of the second and third clauses we have everything we need to translate them into Coq. In Listing 8 we prove that for every function `mu : B -> B` then `relabeling mu` is a blank node mapping. The function `relabeling_term` was defined aiming to meet the blank node mapping's specification. This is why almost every proof in Listing 8 can be proved by computation. The exception is `bnode_to_bnode`, for which we need to proceed by case analysis.

To reason about blank nodes we define `get_b_term` and `get_bs` in Listing 9. These two functions together allow us to project the blank nodes wrapped with a `term`'s constructor. When we apply `get_bs` to a sequence of terms the result is a filtered sequence that contains all the blank nodes `b : B`, which were a `Bnode b` in the input list. Moreover, if we apply `get_bs` to a set of terms we get as result the set of blank nodes of that set. This will help us to define a restricted domain where the RDF isomorphism would have to be bijective (more on this in part III). In this chapter we are just defining the foundation that will allow us to reason about the RDF model.

```
Section blank_node_mapping.
    Variables (I B L : Type).
    Variables (mu : B -> B).

    Definition is_bnode (t : term I B L) :=
      match t with | Bnode _ => true | _ => false end.

    Lemma bnodes_to_bnodes (t : term I B L) :
      is_bnode t -> is_bnode (relabeling_term mu t).
    Proof. by case t. Qed.

    Lemma relabeling_lit l : (relabeling_term mu (Lit l)) = Lit l.
    Proof. by []. Qed.

    Lemma relabeling_iri l : (relabeling_term mu (Iri i)) = Iri i.
    Proof. by []. Qed.

End blank_node_mapping.
```

Listing 8: For every function `mu`, `relabeling_term mu` is a blank node mapping.

```
Definition get_b_term (t : (term I B L)) : option B :=
  if t is Bnode b then Some b else None.

Definition get_bs (ts : seq (term I B L)) : seq B :=
  pmap get_b_term ts.
```

Listing 9: Functions to project blank nodes from a list of terms.

4.1.5 Illustrating terms

To better illustrate we instantiate our model with example types. In Listing 10 we define the type of IRIs to be strings, the type of blank nodes to be the natural numbers, and the type of literals as pairs of strings, where the first component refers to the datatype of the literal, and the second one refers to the literal value. The postfix `%type` indicates Coq to interpret the definition as the product of types, instead of the natural numbers multiplication. Note that this instantiation of term's type allows the construction of ill-formed terms; for example, it is possible to construct a literal whose datatype is `"hello"`. We could be more precise about the structure of literals at the cost of additional definitions and complexity. However, the choice of these types for IRIs, blank nodes, and literals, allows us to illustrate naturally, and closer to a standard RDF setting. The last definition in Listing 10 is `mu`, a mapping from blank nodes to blank nodes; if `mu` receives `0` it assigns it to the blank node `1`, otherwise it assigns it to the same blank node `mu` received as input[1]. The reader can assume that the types `iri_t`, `bn_t`, and `lit_t`, have a declared instance of an `eqType`. We omit the transferring of `eqType` to these types for brevity.

```
Definition iri_t := string.
Definition bn_t := nat.
Definition lit_t := (string * string)%type.

Definition mu (b : bn_t) : bn_t :=
match b with
| 0 => 1
| _ => b
end.
```

Listing 10: Definition of an example types of IRIs, blank nodes, and literals, and an example mapping.

With the definitions in Listing 10 we explain how to define some terms, how to compare them, and relabel them using `CoqRDF`. We will keep using these definitions as we progress in the chapter. In Listing 11 we start by defining some aliases for our term constructors, which helps us improve the readability of the example. We define `I`, `Bn` and `L`, aliases for the constructors of IRIs, blank nodes and literals, respectively; these aliases yield terms of type `term iri_t bn_t lit_t`. Next, we present three examples using equality and relabeling. In the first one, we compare an IRI `I "isA"` against the relabeling of that same IRI. Coq is enabled to compute through the relabeling and recognize the results are equal. In the second example we compare the relabeling of a literal against an IRI representing the same value, proving that they are different. In the last example we show how the relabeling of the blank node `Bn 0` under `mu` yields the blank node `Bn 1`.

We explore the interaction between the components we have defined so far. As a result of that we obtain a library of term's identities. The library is divided in sections, which are organized by theories depending on the assumptions we make about the structure of the set of IRIs, blank nodes and literals. In Listing 12 we display a few lemmas from the library. The

[1]Indicated with an underscore in the second branch of the match.

```
Definition I  := @Iri iri_t bn_t lit_t.
Definition Bn := @Bnode iri_t bn_t lit_t.
Definition L  := @Lit iri_t bn_t lit_t.

Example relabeling_an_iri :
  (relabeling_term mu (I "isA")) == (I "isA") = true. by []. Qed.

Example relabeling_a_literal :
  (relabeling_term mu (L ("number","1781"))) == (I "1781") = false. by []. Qed.

Example relabeling_a_blank_node :
  (relabeling_term mu (Bn 0)) == (Bn 1) = true. by []. Qed.
```

Listing 11: Examples of equality and relabeling at the level of terms.

interested reader can find all the available lemmas in the file *Term.v* from our GitHub repository. The polymorphic properties we formalize describe how `relabeling_term` behaves as a function. We present how `relabeling_term` interacts with the identity map, composition, and term's kind preservation. We find that for every injective map `mu : B1 -> B2`, then `relabeling mu` is also injective, and that it computes in an extensional manner, meaning that if two maps `mu1 mu2 : B1 -> B2` give the same blank nodes, then `relabeling mu1` and `relabeling mu2` will yield the same terms. Regarding the properties about terms with equality, we formalize identities about membership to sequences of terms, and properties, which are preserved under relabeling such as being a blank node or the count of blank nodes in sequences.

```
Section PolyTerm.
  Variables I B B1 B2 B3 L : Type.

  Lemma relabeling_term_id (trm : term I B L) :
    relabeling_term id trm = trm.

  Lemma relabeling_term_comp trm (mu1: B1 -> B2) (mu2 : B2 -> B3) :
    relabeling_term (mu2 \o mu1) trm =
      relabeling_term mu2 (relabeling_term mu1 trm).

  Lemma relabeling_term_ext (mu1 mu2 : B1 -> B2) :
    mu1 =1 mu2 -> relabeling_term mu1 =1 relabeling_term mu2.

  Lemma relabeling_term_inj (mu: B1 -> B2) (inj_mu : injective mu) :
    injective (relabeling_term mu).

End PolyTerm.
Section EqTerm.
  Variables I B L : eqType.

  Lemma bnode_memP (b : B) trms : Bnode b \in trms = (b \in get_bs trms).

  Lemma get_bs_map s mu: all (@is_bnode I B L) s ->
    (get_bs (map (relabeling_term mu) s)) = map mu (get_bs s).

  Lemma all_bnodes_uniq_bs s : all (@is_bnode I B L) s ->
    uniq (get_bs s) = uniq s.

  Lemma undup_get_bs (s : seq (term I B L)) :
    (undup (get_bs s)) = (get_bs (undup s)).

  Lemma mem_get_bs_undup (s: seq (term I B L)) :
    get_bs (undup s) =i get_bs s.

  Lemma get_bs_nil_all_not_b ts :
    reflect (get_bs ts = [::]) (all (fun t=> negb (is_bnode t)) ts).

End EqTerm.
```

Listing 12: A few lemmas for terms with relabeling from the `CoqRDF` library.

4.1.6 Summary

We have presented a mechanized definition of RDF terms in Coq. We described this component with the inductive type `term`, for which we derived math-comp's structures that allow us to deal with equality, choice, countability and partial order. Aiming towards formalizing RDF isomorphism we defined the notion of blank node mappings. We then define `relabeling_term`, to adapt the interface of a function between blank nodes to a function between terms. Afterwards, we specified the conditions for a map `mu` in which `relabeling_term mu` meets the blank node mapping's specification. Subsequently, we developed abstractions to reason about sequences of blank nodes. Next, we present an example defining concrete types of IRIs, blank nodes, literals, and mappings, which we use to instantiate, relabel, and pose propositions about `CoqRDF` terms. We end the term's subsection by developing a library covering the term's polymorphic identities, as well as properties which can be derived when terms are equipped with equality.

In the next section we define RDF triples, and describe some properties of our definition of triples that have an impact when reasoning about triple's equality. Subsequently, we define an encoding for equipping triples with math-comp's structures. We extend the notion of relabeling to triples, and also define abstractions to reason about the triple's terms and blank nodes set. At the end we develop a library for reasoning about RDF triples. We extend the term's polymorphic identities, and add lemmas about the interaction between our previously defined concepts and the new abstractions. Furthermore, we explore these interactions in contexts with different assumptions about terms, but mainly when they have an equality operator.

4.2 `CoqRDF` Triples

In this section we present an implementation of RDF triples in Coq. We model them to be well-formed by construction. Following the structure of the previous chapter, we start by giving an encoding for triples and equipping them with math-comp's canonical structures; we provide some details about equality, and then we extend previously defined notions up to the scope of triples. We also give mechanical specifications to characterize our definitions. Next, we continue illustrating how to define concrete triples, and propositions about them, using `CoqRDF`. We end by exploring how these notions interact together, leaving as a result a mechanized library to reason about RDF triples.

4.2.1 Triples definition

RDF triples were introduced in Section 3.3 as 3-tuples of RDF terms. The triple's terms have to satisfy the following properties: the subject has to be an IRI or a blank node; the predicate an IRI; and the object an IRI, blank node or literal. We reflect this in our triple's definition, which is presented in Listing 13. We define *RDF triples* as a record. As per terms, triples are parameterized by three types: the type of IRIs, blank nodes and literals. Triples have five fields: three terms and two proofs. The three terms account for the subject, predicate and object, and the propositions require that the subject is an IRI or a blank node, and that the predicate is an IRI. Note that there is no need to add a proposition for the object, as terms are *always* an IRI, blank node or literal, by definition, as we illustrate in Listing 14. The way we define triples makes checking its well-formedness trivial. Every triple carries the

```
Record triple (I B L : Type) :=
mkTriple
  { subject : (term I B L)
  ; predicate : (term I B L)
  ; object: (term I B L)
  ; subject_in_IB: is_in_ib subject
  ; predicate_in_I: is_in_i predicate
  }.
```

Listing 13: RDF triples defined as a record.

```
Remark term_def (I B L : Type) (t : term I B L) :
  is_iri t || is_bnode t || is_lit t.
Proof. by case t. Qed.

Remark triple_spec_wf t :
  is_in_ib (subject t) && is_in_i (predicate t) && is_in_ibl (object t).
Proof. by case t=> ? ? []? /= -> ->. Qed.
```

Listing 14: A term is either an IRI, a blank node, or a literal; and a proof that the triples are well-formed.

proofs accounting for its own well-formedness; therefore to check that a triple is well-formed it suffices just to project the proofs carried by every triple; we do so in Listing 14.

4.2.2 Triples equality

Our well-formed by construction design comes with a cost. To show that two records are propositionally equal, one has to show the equality of each of its fields. Triples, as we defined them, carry proofs. It would become technical if we had to show equality of proofs. Luckily as explained in Subsection 2.4.5, when a proof comes from a decision procedure, then two proofs of the same proposition are always equal. Then, to conclude that two triples are the same, it suffices to show just that their subjects, predicates and objects are equal. We need this property formalized in order to define triples' encoding. We do so in Listing 15, where we deal with the proof's equality using proof irrelevance.

4.2.3 Triples encoding

With the purpose of equipping our triples with math-comp's infrastructure we define an encoding for triples. In Listing 16 we define how to encode a triple `tr : triple I B L` into the `term I B L * term I B L * term I B L` type. Similarly, we define a decoding to restore such a triple, as well as a lemma stating that the encoding and decoding cancels out. An important remark about these definitions is how we handle the carried proofs.

```
Lemma triple_inj (I B L : Type) (t1 t2 : triple I B L) :
  subject t1 = subject t2 ->
  predicate t1 = predicate t2 ->
  object t1 = object t2 ->
  t1 = t2.
Proof.
  case: t1 t2 => [s1 p1 o1 sin1 pin1] [s2 p2 o2 sin2 pin2] /= seq peq oeq.
  subst; congr mkTriple; exact: eq_irrelevance.
Qed.
```

Listing 15: To conclude triples' equality it suffices to show the equality of its subjects, predicates and objects.

When encoding the triple we forget the proofs, and when decoding, since not every 3-tuple of terms is a triple, we have to check whether the triple's conditions are satisfied. Then, when proving that these definitions cancel out we take advantage of a property of functions. The predicate behind triple's proofs is decided with a function, and thus, it gives the same result when applied to the same arguments. As we encoded a triple whose terms satisfied the well-formedness conditions, when we decode it back to a triple, its terms will also satisfy those conditions. We then use those definitions to transfer math-comp's structures.

Having the triples defined and equipped with math-comp structures, we proceed to declare how to extract triple's terms and blank nodes. A triple's terms are the set of terms used in a statement. Therefore, being a member of a triple means to belong to that set of terms. We formalize those notions in Listing 17. We define the triple's terms set (`terms_triple`) as the duplicate-free sequence of the subject, predicate and object. We extend that definition to reason about blank nodes (`bnodes_triple`); we filter the terms set using the `is_bnode` predicate. Math-comp defines abstractions to work with membership, which can be transferred to `triple` by declaring the predicate which corresponds to membership. We declare that predicate to be sequences' membership of the triple's terms set. This also allows us to access math-comp's `_ \in _` notation. For stating that "a term `trm` is in the set of terms of the triple `t`", we write `trm \in t`. It is worth mentioning that `trm \in t` works because the inductive `term` is already equipped with an `eqType`. Coq's type inference system is able to use the term's equality predicate to decide membership. Having the `_ \in _` notation gives us infrastructure to work with membership and a concise syntax to express it. If a term belongs to a triple, it can bring us useful equations to work with; for example let us say we have `trm \in t`, then `trm` is either the subject, predicate or object of the triple as we present in Listing 18.

4.2.4 Relabeling triples

To formalize the isomorphism relation we need to extend our notion of relabeling for triples, and eventually, when defining graphs, extend the notion for them as well. We define the relabeling of a triple under a map as the triple resulting after relabeling every term under that map; the mechanized implementation is presented in Listing 20. As every triple is well-

```
  Section CodeTriple.

  Variables (I B L : Type).

  Implicit Type tr : triple I B L.

  Definition code_triple tr : (term I B L * term I B L * term I B L)%type :=
    let: mkTriple s p o  _ _ := tr in (s, p, o).

  Definition decode_triple (t : (term I B L * term I B L * term I B L)%type) :=
    let: (s, p, o) := t in
    if insub s : {? x | is_in_ib x} is Some ss then
      if insub p : {? x | is_in_i x} is Some pp then
        Some (mkTriple o (valP ss) (valP pp))
      else None
    else None.

  Lemma pcancel_code_decode : pcancel code_triple decode_triple.
  Proof.
    case=> s p o ibs ip /=.
    case: insubP=> [u uP us |]; last by rewrite ibs.
    case: insubP=> [v vP vs |]; last by rewrite ip.
    by congr Some; apply: triple_inj.
  Qed.

End CodeTriple.
```

Listing 16: Triples encoding.

```
Definition terms_triple (I B L: eqType) (t : triple I B L) : seq (term I B L) :=
  let (s,p,o,_,_) := t in undup [:: s ; p ; o].

Definition bnodes_triple (t : triple I B L) : seq (term I B L) :=
  filter (@is_bnode I B L) (terms_triple t).

Canonical triple_predType (I B L : eqType):=
  PredType (pred_of_seq \o (@terms_triple I B L)).
```

Listing 17: Functions to extract terms and blank nodes from a triple, and a predicate type declaration.

```
Lemma mem_triple_terms (I B L : eqType) (trm : term I B L) (t : triple I B L) :
  trm \in t = [|| (trm == (subject t)),
                 (trm == (predicate t)) |
                 (trm == (object t))].
```

Listing 18: The specification of term's membership to a triple.

formed by construction, we must prove that the resulting triple preserves its well-formedness. It is then left to prove that the subject remains as an IRI or blank node, which we do by applying the lemma `relabeling_term_preserves_is_in_ib`; and that the predicate is still an IRI, which we prove by applying the lemma `relabeling_term_preserves_is_in_i`; both lemmas are available in our GitHub repository. The notion of relabeling we have just defined differs a bit from the last clause of the RDF isomorphism (3.1), the latter does not relabel the predicate: *The triple (`s`, `p`, `o`) is in G if and only if the triple (`M(s)`, `p`, `M(o)`) is in G'.* As relabeling does not change the IRI contained by the term, we consider that it is reasonable to relabel the predicate as well; it will only update the term's blank node type. That decision forces triples to remain homogeneous with respect to the IRIs', blank nodes' and literals' type, which gives rise to a question: when modeling resources, do the terms used within an RDF graph come from the same set of IRIs, blank nodes and literals? The sets of IRIs, and literals can be considered universal since they are defined extensionally by the standard, but for the case of blank nodes this is not specified. Since we can not compare terms indexed by different blank nodes because of type constraints, we will consider that the set of blank nodes is also universal.

4.2.5 Illustrating triples

Continuing with the example terms that we defined in the previous section, we define example triples using such terms. In Listing 19 we define an alias for the constructor of triples `mkT`, which instantiate the type of IRIs, blank nodes, and literals with the types `iri_t`, `bn_t`, and `lit_t`, respectively (see Listing 10). We also define an alias for the type of triples instantiated with `iri_t`, `bn_t`, and `lit_t`. Next, we declare three triples, each named with the statement it represents; for example, the triple `z_isA_sonata` represents the triple with the blank node

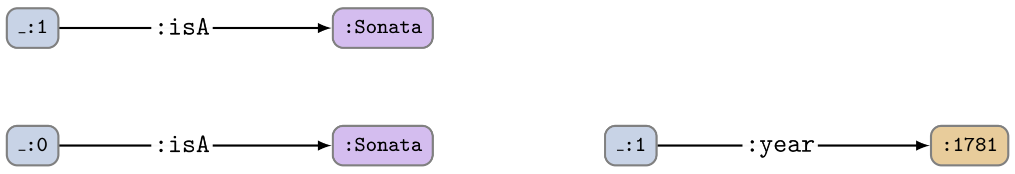


Figure 4.1: Visual representation of the triples defined in Listing 19.

`Bn 0` as the subject, the IRI `I "isA"` as the predicate, and the IRI `I "sonata"` as the object. The tactic `refine` generates one subgoal for every underscore in its argument, where the type of the generated subgoal is the type of the argument where the underscore is placed. In the case of `z_isA_sonata`, `refine` opens two goals, one of type `is_in_ib (Bn 0)` and the other one of type `is_in_i (I "isA")`. Both subgoals are solved by computation by prepending `by`. Finally, we present an example exhibiting the equality between `o_is_a_sonata`, and the relabeling of `z_isA_sonata` under `mu`. In Figure 4.1 we visually display the triples defined in Listing 19.

```
Definition mkT := @mkTriple iri_t bn_t lit_t.
Definition tr_t := triple iri_t bn_t lit_t.

Definition z_isA_sonata : tr_t.
  by refine (@mkT (Bn 0) (I "isA") (I "sonata") _ _). Defined.

Definition o_isA_sonata : tr_t.
  by refine (@mkT (Bn 1) (I "isA") (I "sonata") _ _). Defined.

Definition o_year_1781 : tr_t.
  by refine (@mkT (Bn 1) (I "year") (L ("number","1781")) _ _). Defined.

Example relabeling_a_triple_eq :
  relabeling_triple mu z_isA_sonata == o_isA_sonata = true. by []. Qed.
```

Listing 19: Examples of equality and relabeling at the level of triples.

Before defining RDF graphs we explore how triple's relabeling interacts with triple's terms and blank nodes. We extend the term's identities to triples, under polymorphism and equality. We define the notion of *ground triples*: triples that do not contain blank nodes. For example, two lemmas we prove are: that `relabeling_triple` distributes with respect to function composition; and that if a map is injective, then the terms of a relabeled triple are the same as relabeling the terms of a triple. We present a few more lemmas of our library in Listing 21. The complete list can be found in the file *Triples.v* of our GitHub repository.

```
Definition relabeling_triple B1 B2 (mu : B1 -> B2) (t : triple I B1 L)
  : triple I B2 L :=
  let (s, p, o, sin, pin) := t in
  mkTriple (relabeling_term mu o)
    ((iffLR (relabeling_term_preserves_is_in_ib mu s)) sin)
    ((iffLR (relabeling_term_preserves_is_in_i mu p)) pin).
```

Listing 20: The relabeling of a triple under a map.

```
Section PolyTriple.
  Variables I B B1 B2 L : Type.
  Implicit Type t : triple I B L.

  Definition is_ground_triple t : bool :=
    let (s,p,o,_,_) := t in
    ~~ (is_bnode s || is_bnode p || is_bnode o).

  Lemma relabeling_triple_comp (mu1 : B -> B1) (mu2 : B1 -> B2) t :
      relabeling_triple (mu2 \o mu1) t =
        relabeling_triple mu2 (relabeling_triple mu1 t).

  Lemma relabeling_triple_ext mu1 mu2:
    mu1 =1 mu2 -> relabeling_triple mu1 =1 relabeling_triple mu2.

  Lemma ground_triple_relabeling t (mu : B -> B) :
    is_ground_triple t -> relabeling_triple mu t = t.

  Lemma relabeling_triple_inj (mu : B -> B1) (mu_inj : injective mu) :
    injective (@relabeling_triple B B1 mu).

End PolyTriple.
Section EqTriple.
  Variables I B L : eqType.
  Implicit Type t : triple I B L

  Lemma tripleE t1 t2 : t1 == t2 = [&& (subject t1) == (subject t2),
                                      (predicate t1) == (predicate t2) &
                                        (object t1) == (object t2)].

  Lemma terms_relabeled_triple (B1 B2: eqType) (t : triple I B1 L)
    (mu: B1 -> B2) (inj_mu: injective mu) :
      terms_triple (relabeling_triple mu t) =
        map (relabeling_term mu) (terms_triple t).

End EqTriple.
```

Listing 21: A few lemmas for triples with relabeling from the `CoqRDF` library.

4.2.6 Summary

We have presented a definition of RDF triples in Coq. We equipped them with math-comp's structures, thanks to an encoding of triples. Targeting the formalization of isomorphism, we extended the notion of relabeling to triples, and also defined abstractions to reason about the triple's terms and blank nodes set. With those definitions we declare triples as a predicate type, providing infrastructure to reason about triple's membership. Using `CoqRDF` we illustrate how to define triples and state propositions about them. Next, we explored the interactions of triples with relabeling and our newly defined abstractions. These interactions were studied under different assumptions about the term's structure. As a result we obtain a library which allows us to reason formally about triples.

In the next section we present a definition of RDF graphs in Coq. We make a decision on how to model sets, and discuss the implications of our decision with respect to equality. We derive math-comp's canonical structures for RDF graphs and define abstractions to reason about graph projections, such as their sets of terms and blank nodes. The notion of relabeling is extended for graphs, and we develop a library which allows us to reason formally about the RDF model.

4.3 `CoqRDF` Graphs

In this section we present a definition of well-formed RDF graphs. We persevere in our design of making definitions well-formed by construction. This means that every operation with an RDF graph as output is going to require a proof witnessing the output's well-formedness. We include additional lemmas to tackle the challenge posed by well-formed by construction designs. We equip graphs with different math-comp's structures, which allows us to provide the implementation of more graph operations. In particular, we define a set-equality relation for identifying graphs, and explore how this notion interacts with the rest of the RDF's components and abstractions. We illustrate with examples defining RDF graphs, and declaring propositions about them using `CoqRDF`.

4.3.1 RDF graphs definition

RDF graphs were presented in Section 3.3 as sets of RDF triples. We then have to make a choice on how to model them. There are mainly two options to model sets: to work with finite types or to work with duplicate-free sequences; both options are already supported by the math-comp library. The two options demand a different structure from its type parameters, on the one hand, `fintype` would require the type parameters to be equipped with equality, canonical choice and countablilty. On the other hand, duplicate-free sequences only require its type parameters to be equipped with equality. Demanding more structure from the type parameters means the structure has more restrictions when applying our model to concrete types, as in examples in Listings 11 and 19. As discussed in Sections 4.1 and 4.2, the RDF model has a part of its theory that only relies on equality, this is also the case for

RDF graphs. Finite types are capable of modeling sets, but the infrastructure they provide is too low level for our requirements. The use of finite types often introduce unwanted interdependencies, and hence, difficulties when thinking of long term support. On the other hand, sequences are easy to use, and as they are a common and popular choice they enjoy a vast amount of infrastructure, which we can easily transfer to our definition of graphs. Modeling sets with duplicate-free sequences has the downside of having to make definitions and results order-agnostic. Besides that, `fintype`s introduce unwanted interdependencies, and that are internally with duplicate-free sequences; we chose the second option: the duplicate-free sequences, and deal with the theory of permutations when it becomes necessary. Sequences have to deal explicitly with permutations, but it allows us to keep RDF theory within its necessary requirements, instead of adding conditions that are sufficient but not necessary. We model RDF graphs with a record, which we present in Listing 22. RDF graphs are parameterized by three types: `I`, `B` and `L`, as in `term` and `triple`. We declare two fields for the record: `graph`, a sequence of triples; and `ugraph`, a proof stating that the field `graph` is duplicate-free, which we can formalize using the predicate `uniq` from the math-comp's sequences library. Note that `I`, `B` and `L` are required to be `eqType`. Otherwise, we would not be able to decide if an element is duplicated, and hence, as our definitions are well-formed by construction, a graph definition parameterized by mere types would not suffice to enforce the graph's well-formedness notion.

```
Record rdf_graph (I B L : eqType) :=
  mkRdfGraph {
    graph :> seq (triple I B L) ;
    ugraph : uniq graph
  }.
```

Listing 22: RDF graphs definition in Coq.

4.3.2 RDF graphs encoding

RDF graphs are proof carriers (see 2.4.5), but proof irrelevance (see 2.4.5) also applies in this context because `uniq` is a boolean predicate. We apply the same strategy used for triples to encode graphs, proving `graph_inj`, a lemma stating that two RDF graphs are equal if their sequences are equal. In Listing 23 we define our encoding as projecting the sequence from the graph. The decoding consists in checking that the sequence is duplicate free. As we have been doing in this chapter, we use these functions to transfer math-comp's structures to graphs.

4.3.3 RDF graphs equality

We have declared RDF graphs as an `eqType`, meaning we have an equality operator for our structure. However, this operator does not match the notion of equality we want to capture. Let us explain with an example. Let us have two different triples `t1 t2: triple I B L`, and two RDF graphs `G` and `H` defined as follows `G := mkRdfGraph [:: t1; t2] _.`, and

```
Section CodeRdf.
  Variables I B L : eqType.

  Definition code_rdf g : (seq (triple I B L))%type :=
    graph g.

  Definition decode_rdf (s: seq (triple I B L)) : option (rdf_graph I B L) :=
    if insub s : {? x | uniq x} is Some us
    then Some (mkRdfGraph (valP us))
    else None.

  Lemma pcancel_code_decode : pcancel code_rdf decode_rdf.
  Proof. case=> g ug=> /= ; rewrite /decode_rdf.
         case: insubP=> [? _ ?|]; last by rewrite ug.
         by congr Some; apply: rdf_inj.
  Qed.

End CodeRdf.
```

Listing 23: Graph's encoding and decoding definition.

`H := mkRdfGraph [:: t2; t1] _`. The graphs `G` and `H` are built with the same pair of triples, they are placed in a different order on the graph's sequences. Testing for equality between these two graphs (`G == H`) would result to be `false`, even if `G` and `H` are the same graph; remember that they are defined as sets. The way we have defined them, RDF graphs are using sequence equality as their equality operator, that is, pointwise equality. This is indeed the predicate that matches our graph's propositional equality, but does not coincide with the structure we are trying to model. Having this mismatch between propositional equality and RDF graph's equality poses a challenge. We will have to define an internal operation to reason about RDF graphs which are same. This operation corresponds to set equality, since RDF graphs are defined as triples set. We are looking for a predicate `eqb_rdf`, which relaxes sequence's equality by accepting sequences with the same items, that would be, sequence equality up to permutation, which math-comp defines as `perm_eq`. Math-comp's definition has an additional restriction; `perm_eq` is sensitive to the repetition of items. However, because of `ugraph` we already know that every item in a graph appears just once. i.e. we do not have to worry about this additional requirement, so `perm_eq` fits perfectly for modeling RDF graphs' equality. Indeed, in Listing 24 we prove that two graphs have the same elements if and only if they are sequence-equal up to permutation. Math-comp's definition of equality up to permutation provides us with good infrastructure to reason about the permutation theory of sequences. In particular, `perm_eq` is proved to be an equivalence relation, i.e. reflexive, symmetric and transitive, which give us a good starting point for our graph's notion of equality.

```
Fact rdf_perm_mem_eq {I B L : eqType} (g1 g2 :rdf_graph I B L) :
  (perm_eq g1 g2) <-> (g1 =i g2).
```

Listing 24: Two graphs have the same elements if and only if their sequences are equal up to permutation.

4.3.4 RDF projections

We now define the abstractions to reason about graph's terms and blank nodes. A design issue arises from our intentions. Depending on the level we choose to define these abstractions we are going to have different amounts of technical difficulties when stating lemmas about them. Let us explain with an example. In Listing 25 we present two ways of extracting graph's terms: the first one defined at the level of sequences of triples, and the second one defined at the level of graphs. On whichever level we decide to define it, the abstraction can be implemented similarly: by extracting the terms of every triple in the graph, and then merging the results while removing the duplicated terms. Both implementations `terms_ts` and `terms` look very similar, and indeed they compute the same results. However, let us compare visually how formalizing a property looks for these two definitions. For our example we are using a property stating how to deal with terms when the underlying sequence of our graph is a `cons`. The two lemmas `terms_ts_cons` and `terms_cons` state the same property: they state how the abstraction of graph's terms interacts with `cons` and sequences concatenation. The lemma `terms_ts_cons` expresses information about how terms, cons, and concatenation interact. The lemma `terms_cons` also accounts for that, and, although irrelevant for the operation, also introduces information that graphs are duplicate free. As can be appreciated in Listing 25, `terms_cons` requires the proof `us`, which we need to handle along the statement; for example, since RDF graphs require a proof of its well-formedness, we have to use `uniq_tail : (uniq (a :: t)) -> uniq t` to define a proper RDF graph on the right-hand side of the equation. As a result, the formal statement of `terms_cons` gets polluted with unnecessary details about RDF graphs; and worse, the effort of handling the graph well-formedness proof is needless, as we do not use it in the proof when extracting the terms, and the proof is just dropped in the end. To use this operation with graphs, we have to define a coercion from RDF graphs to sequences of triples, which we declare as the projection of the `graph` field. By dropping the `ugraph` proof we allow the operation to compute further, and to ease the context during proofs. This same reasoning applies to the abstractions of the graph's set of blank nodes, which we define in Listing 26 under the name `bnodes_ts`. We also define `get_bts`, a version of `bnodes_ts` which removes the term's abstraction and allows us to reason about blank nodes directly. We link the two versions with the lemma `bnodes_map_get_bts`, which states that the sequences produced by `bnodes_ts` and `get_bts` are equal, modulo the blank node's constructor `Bnode`.

```
Section terms_cons.
  Variables I B L : eqType.

  Definition terms_ts (ts : seq (triple I B L)) : seq (term I B L) :=
    undup (flatten (map (@terms_triple I B L) ts)).

  Definition terms (g : rdf_graph I B L) : seq (term I B L) :=
    undup (flatten (map (@terms_triple I B L) (graph g))).

  Lemma terms_ts_cons (trpl : triple I B L) (ts : seq (triple I B L)) :
    terms_ts (trpl :: ts) = undup (terms_triple trpl ++ (terms_ts ts)).

  Lemma terms_cons (trpl : triple I B L)
    (ts : seq (triple I B L)) (us : uniq (trpl :: ts)) :
      terms (mkRdfGraph us) =
        undup (terms_triple trpl ++ (terms (mkRdfGraph (uniq_tail us)))).

End terms_cons.
```

Listing 25: Comparing the definition terms projection defined at two different levels of abstraction.

```
Section bnodes_cons.
  Variables I B L : eqType.

  Definition bnodes_ts (ts : seq (triple i b l)) : seq (term i b l) :=
  undup (filter (@is_bnode i b l) (terms_ts ts)).

  Lemma bnodes_ts_cons (trpl : triple I B L) (ts : seq (triple I B L)) :
    bnodes_ts (trpl :: ts) = undup ((bnodes_triple trpl) ++ (bnodes_ts ts)).

  Definition get_bts (ts : seq (triple i b l)) : seq b :=
    get_bs (bnodes_ts ts).

  Lemma bnodes_map_get_bts ts : bnodes_ts ts = map (fun b=> Bnode b) (get_bts ts).

End bnodes_cons.
```

Listing 26: Abstraction to reason about graph's set of blank nodes.

4.3.5 RDF graph relabeling

There are properties, especially when thinking about relabeling graphs, that do need graph's well-formedness proof. However, when that is the case, it is also convenient to show those

```
Definition relabeling_seq_triple
  (I B1 B2 L: Type) (mu : B1 -> B2)
  (ts : seq (triple I B1 L)) : seq (triple I B2 L) :=
  map (relabeling_triple mu) ts.

Definition relabeling
  (I B1 B2 L: eqType) (mu : B1 -> B2)
  (g : rdf_graph I B1 L)  (urel : uniq (relabeling_seq_triple mu (graph g)))
  : rdf_graph I B2 L :=
  @mkRdfGraph I B2 L (relabeling_seq_triple mu (graph g)) urel.
```

Listing 27: Graph's relabeling definition

things separately, meaning that we can separate the two things: the property we want to prove, and that making use of such property preserves the graph's well-formedness. By composing those two requirements we then define RDF graph relabeling. We first define how to operationally perform a relabeling; presented under the name `relabeling_seq_triple` in Listing 27, and ask for a proof witnessing that well-formedness is preserved, as we do in the definition `relabeling`. Note that in the definition of `relabeling` we parameterize the proof of the graph well-formedness. We cannot know beforehand which argument is supporting the proof. Indeed, depending on the mapping used to relabel a graph the proof can vary. For example, if the mapping is injective the result would preserve the lack of duplicates because of that fact. Another example is the case of mapping two different blank nodes to the same blank node, which may generate a duplicated triple. This is completely fine if we remove one of the duplicates; then the argument would be that we took care of the duplication we created by removing it afterwards. These two examples provide a well-formed graph as a result. Moreover, although removing duplicates would always serve as an argument, it does not serve our purpose of formalizing isomorphism, because an isomorphism maps blank nodes in a bijective manner, and hence they do not create duplicates. As it can be useful for other applications we define a constructor for RDF graphs where duplicates are removed automatically, which we name `mkRdf`.

4.3.6 Illustrating RDF graphs

Continuing with the examples we defined in the previous sections, we now define a few RDF graphs using the `mkRdf` graph constructor. In Listing 28 we define four RDF graphs, each one with two triples. First, we declare two graphs `OSonataG` and `OSonataG_perm` whose underlying sequence are set-equal. Both `OSonataG` and `OSonataG_perm` state that `_:1` is a sonata composed in `1781`. These two graphs are compared in the example `graph_eq` stating that they are set-equal, a proof which is followed by computation. We declare two more graphs `ZSonataG` and `relabeled_Z`, the former states that `_:0` is a sonata, and `_:1` was composed in `1781`, and the latter is the graph `ZSonataG` relabeled under `mu`. The two final examples prove that `relabeled_Z` and `OSonataG` are set-equal, and that `ZSonataG` is different from the empty graph. In Figure 28 we present the visual representation of the graphs we defined in Listing 28. On the left side are `OSonataG` at the top and `OSonataG_perm` at the bottom. At the right side we display `ZSonataG` and `relabeled_Z`, the former at the top, and the latter at the bottom.

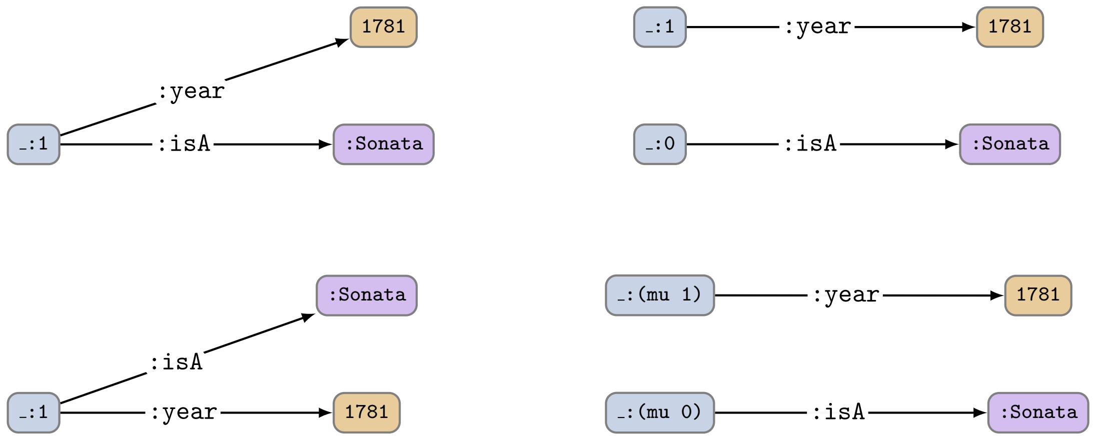


Figure 4.2: Visual representation of the graphs defined in Listing 28

```
Definition OSonataG := mkRdf [:: o_isA_sonata; o_year_1781].
Definition OSonataG_perm := mkRdf [:: o_year_1781 ; o_isA_sonata ].
Definition ZSonataG := mkRdf [:: z_isA_sonata; o_year_1781].
Definition relabeled_Z := (relabeling_undup mu ZSonataG).

Example graphs_eq :
  eqb_rdf OSonataG OSonataG_perm = true. by []. Qed.

Example graphs_relabel_eq :
  eqb_rdf relabeled_Z OSonataG = true. by []. Qed.

Example graphs_prop_neq :
  eqb_rdf ZSonataG (mkRdf [::]) = false. by []. Qed.
```

Listing 28: Examples of equality and relabeling at the level of RDF graphs.

Finally, we explore the interaction between relabeling and blank nodes. Particularly, we study how membership and mapping's properties behave around `bnodes_ts` and `relabeling`. We list a few lemmas about these interactions in Listing 29. For a complete view of the lemmas we proved we refer the reader to the file *Rdf.v* from the repository.

4.4 Summary

In this section we presented a characterization of well-formed RDF graphs defined in Coq. Following math-comp's workflow, we derive canonical structures for RDF graphs. We point out the difference between the propositional equality of our definition and the one expected for RDF graphs. We then propose a predicate to decide graph equality which satisfies RDF's specification. We also propose abstractions to project graph's terms and blank nodes. Subsequently, the relabeling notion is extended to RDF graphs. Next, we illustrate using `CoqRDF` to define graphs, and declare propositions about them. Finally, we explore the interaction between the abstractions and definitions we have declared.

In the chapter we defined RDF terms, triples and graphs following a well-formed by construction design. We equipped each RDF component with several of math-comp's canonical instances. Towards formalizing the notion of isomorphism we defined the notion of a blank node mapping, and developed, in Coq, a mechanized definition meeting that specification, which we named relabeling. We extended the notion of relabeling to provide an interface for every component in the RDF model. For every abstraction we define, we make sure to preserve well-formedness. We illustrated with examples using `CoqRDF`, where we declare concrete terms, triples, and RDF graphs, and made propositions about them. With our implementation we have covered a mechanized definition of the RDF model, and its core abstractions, which can be used to reason about its components.

In the next part we dive into the notion of *isomorphism*, presenting how this notion behaves within a simpler structure, namely directed graphs. That gives us a clear understanding of what to expect from isomorphism at the level of RDF. We then tackle the formalization of RDF isomorphism by identifying how the different aspects of isomorphism are composed. This approach allows us to gradually reach a mechanical formalization of RDF isomorphism. We then apply our implementation to a special kind of algorithm for RDF graphs: *iso-canonical mappings*. These algorithms decide which graph represents all the other graphs within the same isomorphism class. We provide an implementation of an iso-canonical map, and study the structural assumptions that suffice to prove the iso-canonical mapping's specification.

```
Variables I B B1 B2 : eqType.
Implicit Type g : rdf_graph I B L.

Lemma terms_ts_relabeled_mem (ts : seq (triple I B1 L)) (mu: B1 -> B2) :
  (terms_ts (relabeling_seq_triple mu ts)) =i
    (map (relabeling_term mu) (terms_ts ts)).

Lemma bnodes_ts_relabel_inj (ts: seq (triple I B1 L)) (mu: B1 -> B2)
  (mu_inj : injective mu):
    bnodes_ts (relabeling_seq_triple mu ts) =
      (map (relabeling_term mu) (bnodes_ts ts)).

Lemma bijective_eqb_rdf mu nu g1 g2 : forall us1 us2,
  cancel mu nu ->
    eqb_rdf (@relabeling _ _ mu g2 us1) g1 ->
      eqb_rdf g2 (@relabeling _ _ nu g1 us2).

Lemma bnodes_map_get_bs ts :
  bnodes_ts ts = map (fun b=> Bnode b) (get_bs (bnodes_ts ts)).

Lemma perm_eq_bnodes_relabel ts1 ts2 mu :
  perm_eq (get_bs (bnodes_ts (relabeling_seq_triple mu ts1)))
          (get_bs (bnodes_ts ts2)) ->
    perm_eq (undup [seq mu i | i <- get_bs (bnodes_ts ts1)])
            (get_bs (bnodes_ts ts2)).

Lemma perm_eq_bnodes_relabel_inj_in ts1 ts2 mu :
  {in (get_bs (bnodes_ts ts1))&, injective mu} ->
    perm_eq (get_bs (bnodes_ts (relabeling_seq_triple mu ts1)))
            (get_bs (bnodes_ts ts2)) ->
      perm_eq [seq mu i | i <- get_bs (bnodes_ts ts1)]
              (get_bs (bnodes_ts ts2)).

Lemma can_bs_can_rtbs ts (mu nu: B -> B) :
  {in get_bs (bnodes_ts ts), nu \o mu =1 id} ->
    {in ts, [eta relabeling_triple (nu \o mu)] =1 id}.

Lemma get_bs_map s mu : all (@is_bnode I B L) s ->
  (@get_bs I B L (map (relabeling_term mu) s)) = map mu (get_bs s).

Lemma can_bs_can_rtbs ts (mu nu: B -> B) :
  {in get_bs (bnodes_ts ts), nu \o mu =1 id} ->
    {in ts, [eta relabeling_triple (nu \o mu)] =1 id}.
```

Listing 29: A few lemmas for graphs with relabeling from the `CoqRDF` library.

Part III

An application: Reasoning about iso-canonical algorithms

In this part we dive into *isomorphism*. In Chapter 5, we start by presenting how isomorphism behaves on a simple structure: *undirected graphs*. This allows us to extrapolate what to expect from isomorphism with respect to RDF graphs. We continue by decomposing RDF isomorphism components and studying them, which leads us to extend `CoqRDF` with a formal and operational definition of RDF isomorphism. We prove that RDF isomorphism is an equivalence relation. First we give a pen and paper proof, which we then formalize in `CoqRDF`. We remark on the proof's similarities, and the challenges of the mechanized formalization of this property.

Going further with isomorphism, we present the notion of *iso-canonical* mappings, and formalize their specification in Coq, in Chapter 6. Afterwards, we present κ*-mapping*, an instance of an iso-canonical mapping, which we then implement. Subsequently, we study what suffices to prove that κ-mapping is correct with respect to our specification of iso-canonical mappings.

Chapter 5

Isomorphism

In this chapter we cover the notion of *isomorphism*. We present the background of *isomorphism* in the graph theory field, in Section 5.1, bringing us a better understanding of isomorphism's roots, which subsequently leads us to describe precisely what properties to expect from RDF isomorphism in Section 5.2. We then decompose RDF isomorphism, modularize the study of its conditions, and formalize them in Coq in Section 5.3. Next, we propose a boolean predicate, deciding whether a particular mapping is an isomorphism between two given RDF graphs. This predicate allows us, first, to study the properties of RDF isomorphism, such as that it is an equivalence relation; and second, it allows us to explore the interaction between isomorphism and graph relabeling. As a result we obtain a mechanically formalized relation of isomorphism for RDF graphs, which we prove to be an equivalence relation, and a formalized library of lemmas about RDF graphs and isomorphisms.

5.1 Graph isomorphism

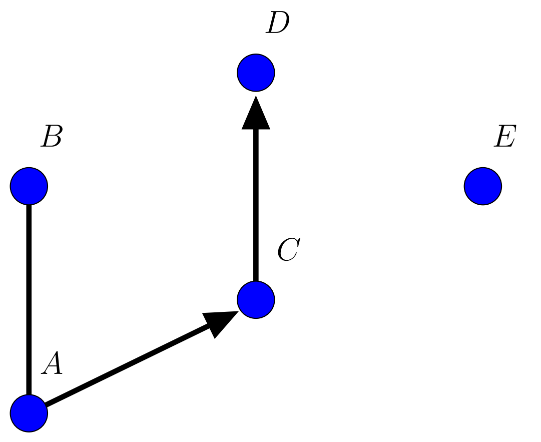
In this section we introduce basic notions of graph theory. The literature has plenty of graph theory definitions, that in general do not differ much, apart from subtleties. Most of the definitions we use come from Diestel's graph theory book [21].

Definition 5.1 *A* directed graph *is a pair of sets (V,E), such that $E \subseteq V \times V$.*

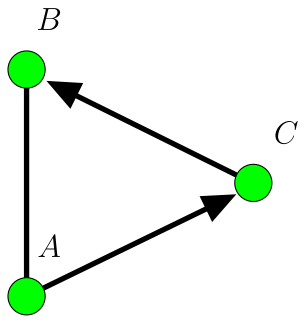
We denote the number of vertices of a graph G with $|G|$. The usual way to draw a graph is to display the $|V|$ vertices, and to connect every pair $(v, u) \in E$ with an arrow from v to u. When the two edges (v, u) and (u, v) are present we draw a line connecting v and u, instead of an arrow. We illustrate with a graph drawing in Figure 5.1a.

Definition 5.2 (Adjacency) *Two vertices x, y of G are adjacent, or neighbors, if and only if (x, y) or (y, x) are an edge of G.*

In the graph in Figure 5.1a, the vertices A and B are adjacent, and the vertex E has no neighbors. Many properties and definitions depend on specific characteristics of the vertices'



(a) Graph on $V := \{A, B, C, D, E\}$ and edge set $E := \{(A, B), (A, C), (B, A), (C, D)\}$.



(b) Graph on $V := \{A, B, C\}$ and edge set $E := \{(A, B), (A, C), (B, A), (C, B)\}$.

Figure 5.1: Two graphs with different structure.

adjacency. For example, checking structure preservation depends on the preservation of adjacency, and a map that preserves structure is called an *isomorphism*; we formalize this notion in Definition 5.3.

Definition 5.3 (Graph isomorphism) *Let $G := (V_G, E_G)$ and $H := (V_H, E_H)$ be two graphs. A map φ is an isomorphism from G to H, if and only if, φ is bijective, and φ and φ^{-1} preserve the adjacency of vertices, that is, $(x, y) \in E_G \iff (\varphi(x), \varphi(y)) \in E_H$ for all $x, y \in V_G$. Two graphs G and G' are isomorphic, noted as $G \simeq G'$, if there exists an isomorphism from G to G'.*

Note that an *isomorphism* may not assign two, or more, vertices to the same result, e.g., the following would not be an isomorphism:

$$\varphi(x) = \begin{cases} B & x \text{ is E or D} \\ x & \text{otherwise} \end{cases}$$

If we apply the map φ above to the graph in Figure 5.1a, we collapse the vertices B, D and E into the vertex B, while preserving adjacency with respect to the result. However, φ is not injective and therefore it cannot be an isomorphism. An isomorphism between two graphs may sometimes be easy to guess, but sometimes it is not. In Figure 5.3 we present an example of the latter. In this example, the isomorphism connecting the graph on the left with the graph on the right is not evident, however these graphs are isomorphic under the identity map. Taking the green vertex for example, we see that on the left side it has three edges, one towards the blue vertex, another towards the pink one and another towards the yellow one, which also happens in the graph on the right. This reasoning can be applied to all nodes exposing the bijective map connecting the two graphs, and which also preserve adjacency, thus the graphs are isomorphic, despite drawing them differently. Note the way structure is preserved with an isomorphism. Vertices and edges are preserved. The first is achieved because the mapping must be bijective, thus the number of nodes is maintained. The second is achieved because the mapping and its inverse must preserve adjacency.

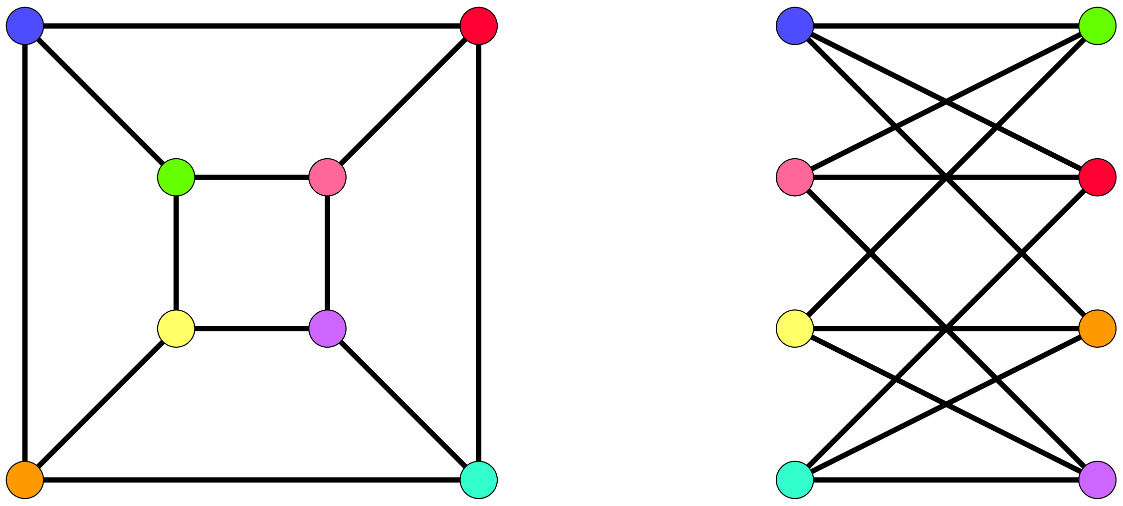


Figure 5.3: Two isomorphic graphs drawn in different ways[1].

Again, many notions arise from isomorphism. For example, there are classes of graphs sharing properties with their isomorphic versions. Also, there are maps that output the same results for isomorphic graphs. We describe these two notions in Definitions 5.4 and 5.5, respectively. An example of a graph property is to contain a triangle. If a graph G contains three pairwise adjacent vertices, then so does every graph isomorphic to G. A graph invariant could be, for example, the number of vertices of a graph. If the number of vertices of a graph G is n, then for every graph G' such that $G \simeq G'$ it holds that $|V_{G'}| = n$.

Definition 5.4 (Graph property) *A class of graphs that is closed under isomorphism is called a graph property.*

Definition 5.5 (Graph invariant) *A map taking graphs as arguments is called a graph invariant if it assigns equal values to isomorphic graphs.*

In this section we have defined basic notions about graphs. These notions are useful to us to understand RDF isomorphism, which we address in the next section. Afterwards, in Chapter 6, we formalize the notion of *iso-canonical algorithms*, i.e., maps that are graph invariant and return graphs that are isomorphic to the input.

5.2 RDF isomorphism

Since RDF is a graph based data model, we may look at RDF definitions through the lense of graphs. Definitions would not fit directly, as RDF has more structure, but they can be adapted. Let us have another look at the definition of RDF isomorphism (Def 3.1). We show again the RDF isomorphism specification in Definition 5.6.

[1]Drawing from the graph isomorphism's Wikipedia article.

Definition 5.6 *Two RDF graphs G and G' are isomorphic (that is, they have an identical form) if and only if there exists a bijection M between the sets of nodes of the two graphs, such that:*

- *M maps blank nodes to blank nodes.*
- *$M(lit) = lit$ for all RDF literals lit which are nodes of G.*
- *$M(iri) = iri$ for all IRIs iri which are nodes of G.*
- *The triple $(s,\ p,\ o)$ is in G if and only if the triple $(M(s),\ p,\ M(o))$ is in G'.*

With this definition, M shows how each blank node in G can be replaced with a new blank node to give G'.

We notice some similarities with respect to the definition of graph isomorphism (Def 5.3), for example, that the mapping must be bijective. At the same time, the fourth clause defines how to check that the adjacency is preserved, and the first three clauses define a particular way of preserving adjacency. This particular manner of preserving adjacency tells us that literals and IRIs are a structural part of RDF, while the blank nodes may vary. As blank nodes represent the existence of a value, changing the blank node does not change the meaning of the term, whereas if we change a literal for another one, it certainly modifies the resource being described.

Having pointed out a few similarities between graph isomorphism and RDF isomorphism, we explore the properties of RDF isomorphism. We focus particularly on proving that it is an equivalence relation (Prop 5.7), i.e. RDF isomorphism is reflexive, symmetric and transitive. We prove these properties in Lemmas 5.8, 5.9 and 5.10, respectively.

Proposition 5.7 *RDF isomorphism is an equivalence relation.*

Lemma 5.8 (RDF isomorphism is reflexive)
For any RDF graph G, $G \simeq G$.

Proof. By the existence of the identity map *id*, which is bijective. The *id* map assigns any term to itself, moreover: it maps blank nodes to blank nodes. In the case of literals, the proof is followed by the definition of *id*. The equation $id(lit) = lit$ holds for all the nodes, hence particularly for the nodes of G. Idem for IRIs. Finally, for any triple (s, p, o) in G, then the triple (id(s), p, id(o)) = (s, p, o) is also in G and vice versa. □

Lemma 5.9 (RDF isomorphism is symmetric)
For all RDF graphs G and H, $G \simeq H \iff H \simeq G$.

Proof. (if) As $G \simeq H$, there exists a bijection M satisfying the four requirements of RDF isomorphism. Since M is bijective, there is M^{-1} canceling the action of M. Then, for every triple (s, p, o) in H it follows that the triple ($M^{-1}(s)$, p, $M^{-1}(o)$) is in G, and since G and H are isomorphic, then every triple in H can be pictured as (M(s), p, M(o)); next as M is bijective and M^{-1} cancels M, it holds that ($M^{-1} \circ M(s)$, p, $M^{-1} \circ M(o)$) = (s, p, o) is in G. The remaining conditions hold by contradiction. If M^{-1} does not map blank nodes to blank nodes, then M did not as well, because M is a bijection. Similarly, if M^{-1} does not map literals and IRIs to themselves, M did not as well. (only if) Idem. $\square$

Lemma 5.10 (RDF isomorphism is transitive)
For all RDF graphs G, H, I, if $G \simeq H$ and $H \simeq I$, then $G \simeq I$.

Proof. Since G is isomorphic to H, and H is isomorphic to I, there exists two maps M_{GH}, and M_{HI}. Then, there exists $M_{HI} \circ M_{GH}$, which is bijective because of composition of bijective functions. It maps the blank nodes of G to blank nodes of I by the definition of composition. The composition of functions mapping blank nodes to blank nodes also map blank nodes to blank nodes, then $M_{HI} \circ M_{GH}$ maps blank nodes to blank nodes. Similarly, for literals and IRIs, as composition of functions mapping literals and IRIs to themselves, then $M_{HI} \circ M_{GH}$ maps literals and IRIs to themselves. $\square$

We have presented what we can expect from RDF isomorphism. We did this by extending the notions of graphs to interpret them from an RDF perspective. Finally, we proved that RDF isomorphism is an equivalence relation. In the next section, we mechanize a formal definition of RDF isomorphism in Coq, and afterwards, we mechanize the proofs of Lemmas 5.8, 5.9, and 5.10.

5.3 `CoqRDF` isomorphism

In this section we mechanize, in Coq, the notion of RDF isomorphism. To do so, we decompose RDF isomorphism by first defining the notion of *pre-isomorphism*, which we study, and note its correspondence with specific aspects of RDF isomorphism. We develop a definition of RDF isomorphism matching the specification of the W3C [18]. We prove that our mechanized definition is an equivalence relation; we study RDF isomorphism further, which results in a library of mechanized lemmas about RDF isomorphism.

```
Section pre_isomorphism.
  Variables I B L : eqType.

  Definition is_pre_iso_ts (ts1 ts2 : seq (triple I B L)) (mu : B -> B) :=
    perm_eq (map mu (get_bts ts1)) (get_bts ts2).

  Definition pre_iso_ts (ts1 ts2 : seq (triple I B L)) :=
    exists (mu : B -> B), is_pre_iso_ts ts1 ts2 mu.

End pre_isomorphism.
```

Listing 30: Pre-isomorphism definition.

5.3.1 Pre-isomorphism

First, note that the definition of RDF isomorphism (Def 5.6) asks for a bijection M between two sets of nodes. Now, remember that if we have a graph (`g : rdf_graph I B L`), then its blank nodes are projected with (`get_bts g`). With the abstractions provided by `CoqRDF`, we are aiming to represent the image of a map for a certain finite-set of inputs. One possibility is to explicitly enumerate the image of the map for the input set that is of our interest. To do that, we make a correspondence between two sequences. The first sequence would be an enumeration of the input set, and the second one the image of the input set under the map. In Coq, we would state this as follows: given two sequences of triples `ts1 ts2` and a map of blank nodes `mu : B -> B`, if `map mu (get_bts ts1) == (get_bts ts2)`, then `mu` maps every element of `get_bts ts1` in a bijective manner. We can establish the bijection locally, because we have a one to one correspondence between two duplicate-free sequences. This models our requirements, but as we are dealing with sets, if the correspondence matched a different permutation of the sequence, the predicate would return an incorrect answer. We correct that by building a predicate which considers the possibility of matching a permutation. We formalize such a predicate in Listing 30 under the name `is_pre_iso_ts`; we also define its propositional version as `pre_iso_ts`. Note that `is_pre_iso_ts` silently captures important properties. For example, it captures that `mu` is injective in the `ts1`'s blank nodes set (i.e. `{in ts1&, injective mu}`); and that `ts1` and `ts2` have the same number of blank nodes. Another benefit of this definition is that it can be used on RDF graphs, because of the coercion we defined from RDF graphs to sequences of triples.

Pre-isomorphism is a relation between two sequences of triples establishing a one-to-one correspondence between its sets of blank nodes. We also say that a map witnessing that relation is a pre-isomorphism. Note that all isomorphisms are pre-isomorphisms. Thus, to prove that isomorphism is an equivalence relation, we prove that `pre_iso_ts` is an equivalence relation as well. In Listing 31 we present the formal statements, whose proofs are available in the `CoqRDF` repository. The proposition `pre_iso_ts` requires an explicit witness certifying that the proposition holds. For reflexivity, we expose the identity. For transitivity, we expose the composition of the given isomorphisms. Hence, these two proofs resemble mostly the pen and paper ones in Lemmas 5.8 and 5.10. For the proof symmetry, it is also the case that it resembles the pen and paper proof, but the challenge posed by its mechanical formalization

```
Lemma pre_iso_ts_refl ts : pre_iso_ts ts ts.
Lemma pre_iso_ts_sym ts1 ts2 : pre_iso_ts ts1 ts2 <-> pre_iso_ts ts2 ts1.
Lemma pre_iso_ts_trans ts1 ts2 ts3 :
  pre_iso_ts ts1 ts2 -> pre_iso_ts ts2 ts3 -> pre_iso_ts ts1 ts3.
```

Listing 31: Pre-isomorphism is an equivalence relation.

is bigger. The issue is that even if there is a map `mu`, such that the sets of blank nodes are in correspondence through `mu`, and that this correspondence is modulo permutation, we cannot extract the inverse directly. This context does not tell us which of the permutations establishes the equality between the two sequences. So, we devise a mechanism to invert a map `mu` modulo permutation. To do that, first, we generate an equality between the two sequences, to be able to invert the map connecting them. Then, we reason without loss of generality. We assume an equality, which we use to prove symmetry, and next, we prove that such equality is true. Thus, given any function `f`, if we assume the equality between `map f (get_bts ts1)` and `get_bts ts2`, we can extract the inverse of `f` through the specification of `map`. Then, we expose a function `g`, such that `map g (get_bts ts1)` and `get_bts ts2` are equal. This function `g` is justified by the existence of the pre-isomorphism between `ts1` and `ts2`, and that both sequences `get_bts ts1`, and `get_bts ts2` are duplicate-free, and have the same size.

Finally, for every map `mu` which is a pre-isomorphism, then `relabeling_term mu` is a blank node mapping, locally bijective in the set of blank nodes of the graph, and meeting the first three clauses of being an RDF isomorphism (because it is a blank node mapping). It is left to formalize a relation which adds the notion of preserving RDF adjacency. In the next section, we extend the notion of pre-isomorphism to give a mechanized formal definition of RDF isomorphism.

5.3.2 Characterizing RDF isomorphism

In this subsection we study adjacency preservation, the last missing piece to mechanize a definition of RDF isomorphism. As we have done through the chapters, we make our definitions on sequences of triples, and then we apply them on RDF graphs, taking advantage of the coercion from RDF graphs to sequences of triples that we have defined. In Listing 32, following the RDF specification [18], we define a predicate to check if a certain map satisfies the conditions of RDF isomorphism. Given two sequences of triples `ts1` and `ts2`, and a map of blank nodes `mu`: if `mu` is a pre-isomorphism between `ts1` and `ts2`, there are two conditions left for `mu` to become an isomorphism. First, that the sequence resulting from the relabeling of `ts1` under `mu` is duplicate-free, this guarantees the well-formedness of the resulting graph; and second, that the result of relabeling `ts1` under `mu` is set-equal to `ts2`, the preservation of adjacency. We present such a predicate in Listing 32, along with its propositional version. Also, note that the appearance of `uniq (relabeling_seq_triple mu ts1)` in the middle clause of `is_iso_ts`, is only required because we are using sequences of triples to define the predicate. When we apply the predicate `is_iso_ts` with RDF graphs as arguments, the clause of well-formedness preservation is subsumed by the preservation of adjacency, as we

```
Section isomorphism.
    Variables I B L : eqType.
    Implicit Type ts : seq (triple I B L).

    Definition is_iso_ts ts1 ts2 mu :=
    [&& is_pre_iso_ts ts1 ts2 mu,
      uniq (relabeling_seq_triple mu ts1) &
        perm_eq (relabeling_seq_triple mu ts1) ts2].

    Definition iso_ts ts1 ts2 := exists mu, @is_iso_ts ts1 ts2 mu.

End isomorphism.
```

Listing 32: RDF isomorphism definition.

```
Lemma perm_eq_relab_uniq g1 g2 mu :
  perm_eq (relabeling_seq_triple mu g1) g2 ->
    uniq (relabeling_seq_triple mu g1).
```

Listing 33: RDF Graphs well-formedness is subsumed by adjacency preservation.

present in Listing 33.

We now prove in Coq that `iso` is an equivalence relation. We present the formal statements in Listing 34, proving that `iso` is reflexive, symmetric, and transitive. Similarly to pre-isomorphism, the proofs of reflexivity and transitivity resemble the pen and paper proofs. For proving that `iso` is reflexive, we show a map satisfying the three conditions of RDF isomorphism. By exposing the identity map as witness, the first clause is exactly `pre_iso_ts_refl` from Listing 31. Then, by `perm_eq_relab_uniq` it suffices to show that adjacency is preserved, which we can conclude, because relabeling a graph under the identity is exactly the same graph, and hence has exactly the same adjacency. For proving transitivity, the first clause is exactly `pre_iso_ts_trans`. To conclude, because of `perm_eq_relab_uniq`, it suffices to show that adjacency is preserved under composition. Again, the proof of symmetry presents a bigger challenge. We have to be cautious and pick the good inverse candidate; not every pre-isomorphism from the blank nodes of an RDF graph G to blank nodes of an RDF graph H is an isomorphism, we provide an example below.

Let us present a concrete example: consider an RDF graph G, composed of two triples: (`_:a`, `:p`, `_:b`) and (`_:c`, `:q`, `_:c`), where `_:a`, `_:b` and `_:c` are different blank nodes, and `:p` and `:q` are different IRIs; we illustrate G in Figure 5.4. Now, let us say we have a pre-

```
Lemma iso_refl g : iso g g.
Lemma iso_sym g1 g2 : iso g1 g2 <-> iso g2 g1.
Lemma iso_trans g1 g2 g3 : iso g1 g2 -> iso g2 g3 -> iso g1 g3.
```

Listing 34: RDF isomorphism is an equivalence relation.

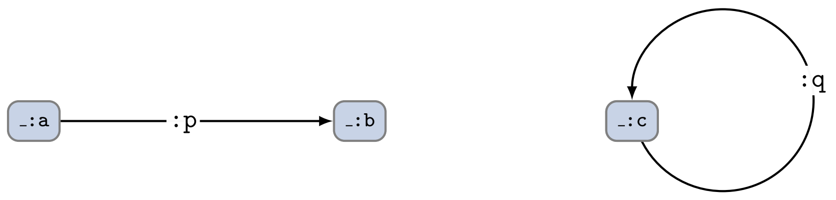


Figure 5.4: RDF graph G with two triples (`_:a`, `:p`, `_:b`) and (`_:c`, `:q`, `_:c`).

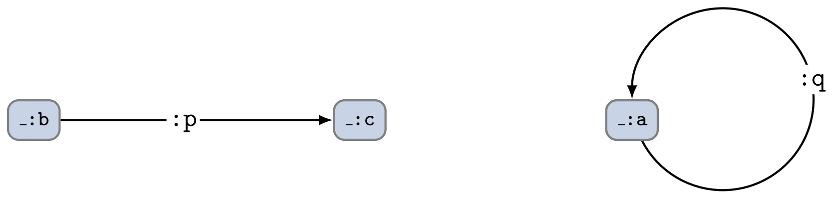


Figure 5.5: RDF graph G' with two triples (`_:b`, `:p`, `_:c`) and (`_:a`, `:q`, `_:a`).

isomorphism mu from G to an RDF graph H, whose set of blank nodes has `_:d`, `_:e` and `_:f`. Next, we choose a particular behavior for mu, and with that behavior we define another pre-isomorphism nu, going in the other direction, from H to G. We define the behaviors of mu and nu in the two equations below.

$$mu(x) = \begin{cases} d & x \text{ is a} \\ e & x \text{ is b} \\ f & x \text{ is c} \\ x & \text{otherwise} \end{cases} \qquad nu(x) = \begin{cases} b & x \text{ is d} \\ c & x \text{ is e} \\ a & x \text{ is f} \\ x & \text{otherwise} \end{cases}$$

Both maps, mu and nu satisfy the conditions of a pre-isomorphism, but considering their particular behavior, when relabeling G under $(nu \circ mu)$, the maps do not cancel. We end with a result which is different from G. Indeed, we obtain a graph G' which has different triples. The triples of G', (`_:b`, `:p`, `_:c`) and (`_:a`, `:q`, `_:a`), describe the resources in the same way as G, but with different blank nodes, as we illustrate in Figure 5.5. The graph G has the blank node `_:c` related to itself under the predicate `:q`, whereas in G' the blank node related to itself under the predicate `:q` is `_:a`. Also, the triple connecting the two different blank nodes also differs, on G, the blank nodes `_:a` and `_:b` are the ones connected by the predicate `:p`, whereas on G', the predicate `:p` connects `_:b` and `_:c`. The graphs G and G' are isomorphic, but we need them to be equal to establish the symmetry of isomorphism.

We see that not every pre-isomorphism going in the opposite direction satisfies the symmetry. Rather, given a pre-isomorphism mu from the blank nodes of an RDF graph G to the blank nodes of another RDF graph H, to prove the symmetry of isomorphism, it suffices to show the existence of a pre-isomorphism nu, from the blank nodes of H to the blank nodes of G, which cancels the action of mu on the blank nodes of G, while preserving adjacency. The mechanized statement we need to satisfy is `{in (get_bts G), nu \o mu =1 id}`, which reads: the composition of mu and nu assigns the same values as the identity, in the particular domain of the blank nodes of G. Using a similar argument as for pre-isomorphism symmetry, we build a pre-isomorphism with such property. Subsequently, we extend the cancelation property of the map nu to the level of triples, meaning that we prove that: for all triples contained in G, relabeling that triple under the composition of mu and nu assigns the same values as the identity, which in mechanized terms corresponds to the following statement: `{in G, [eta relabeling_triple (nu \o mu)] =1 id}`. That extension of the cancelation property allows us to prove the preservation of adjacency, which lets us conclude that `iso` is symmetric.

We have thus provided a formal mechanized definition of RDF isomorphism using `CoqRDF`, which we prove to be an equivalence relation.

5.4 Summary

In this chapter we presented basic notions of graph theory. We dived into the structure of directed graphs, and graph isomorphism. We used these concepts to understand what properties to expect from RDF isomorphism, which we then proved to be an equivalence relation. Afterwards, we mechanize the notion of RDF isomorphism in Coq, using the `CoqRDF` library. We define it gradually, by decomposing the different conditions to satisfy RDF isomorphism. Next, we mechanize our proof that RDF isomorphism is an equivalence relation. We compare these proofs, and remark on the challenge posed by the symmetry of RDF isomorphism.

The possibility of reasoning about *isomorphism* opens the door to reasoning about notions built on top of it, for example, reasoning about the concept of *iso-canonical* algorithms. In the next chapter, we present the concept of *iso-canonical* algorithms, and formalize its mechanized definition, and properties, in `CoqRDF`. After giving a mechanized formal specification of *iso-canonical* algorithms we implement an instance of it, *κ-mapping*. We explain, first, how to implement *κ-mapping* using functional programming; second, the techniques we use to prove that *κ-mapping* returns graphs which are isomorphic to the input, and that two graphs give a set-equal result under κ-mapping, if and only if, the input graphs are isomorphic. These two properties allows us to conclude that κ-mapping is an iso-canonical algorithm.

Chapter 6

Iso-canonical algorithms

In the previous chapter we mechanized in Coq a formal definition of RDF isomorphism: `iso`. Then, we proved that `iso` is an equivalence relation. In this chapter we mechanize a specification built on top of RFD isomorphism, we present the notion of *iso-canonical* algorithms and motivate its applications in Section 6.1. Afterwards we present *κ-mapping*: an instance of an *iso-canonical* algorithm in Section 6.2. We define κ-mapping in a functional setting with a dependent type system, in contrast to its usual imperative presentation in Section 6.3. Subsequently, we study the requirements to certify the correctness of an iso-canonical algorithm implementation. We prove in Coq that our implementation of κ-mapping returns a proper well-formed graph in Section 6.4. We also prove that for any graph, κ-mapping returns a graph which is isomorphic to the input graph in Section 6.5. Finally, we prove that κ-mapping is a graph invariant, which returns a canonical instance of its input in Section 6.6. With the latter two properties we provide a completely mechanized proof that κ-mapping is an iso-canonical algorithm.

6.1 Canonical instances

In Section 3.4, we presented that because RDF syntax parsers can assign blank node identifiers arbitrarily, RDF test suites must check their results modulo isomorphism, rather than under RDF equality. However, the complexity of checking RDF isomorphism (3.1) between two graphs is a *GI-complete* problem, a result proven by Hogan [34]. The complexity class GI groups the algorithms solving graph isomorphism (5.3), being an intermediate class between the polynomial and *NP-complete* classes, that is, GI describes a problem in which a solution in polynomial time is not known, nor is it NP-complete. The latest results about the complexity of GI, proven by Babai [3], show that graph isomorphism can be solved in *quasi-polynomial* time. Babai presented an algorithm solving the graph isomorphism problem in $\exp((\log\ n)^{O(1)})$ time, where n is the number of vertices of the graph. On the other hand, to empirically analyze research results, the RDF community uses the latest Billion Triple challenge (BTC) Dataset. BTC Datasets have been published since 2008 [56, 32], challenging research into improving how to analyze and extract value from RDF data, as well as tracking the adoption of the Semantic Web. The latest BTC dataset that has been published at the time of writing this document corresponds to BTC 2019 [33], and the previous one is BTC 2014 [5]. An analysis of BTC 2014 [34] indicates that the largest RDF document from BTC 2014 has 7.3 million triples and 254,288 blank nodes. Additionally, the average size of RDF documents has increased, the 2014 BTC dataset [5] has an average of 91 statements per document, while the one from 2019 [33, 32] has 816. Due to the complexity of RDF isomorphism, and the increasing size of RDF documents, it becomes necessary to find more efficient ways to check isomorphism between RDF graphs.

When checking isomorphism periodically, there are solutions lowering the cost of consecutively checking isomorphism. These solutions are related to the *canonical instance* of a graph. The canonical instance of a graph G is a graph G_{can} that satisfies two properties: first, that G_{can} is a graph which represents the class of graphs isomorphic to G, and second, that G_{can} is isomorphic to G. Likewise, iso-canonical algorithms are maps assigning graphs to their canonical instance; we formalize this notion in Definition 6.1.

Definition 6.1 (Iso-canonical algorithm) *A map M from RDF graphs to RDF graphs is iso-canonical if and only if:*

1. *For all RDF graph G, it holds that $G \simeq M(G)$, and*
2. *For all RDF graphs G and H, it holds that $G \simeq H \iff M(G) = M(H)$.*

Broadly speaking, iso-canonical algorithms design a way to assign blank node identifiers in a deterministic way. Then, checking isomorphism between two canonical instances is solved in polynomial time, with respect to the number of triples. Finding the canonical instance of a graph has also been proven to belong to the quasi-polynomial complexity class. Babai [4] developed an algorithm constructing a canonical form of a graph within the same time bound, i.e., quasi-polynomial time. So far, the algorithms that construct canonical instances have been built based on algorithms that verify isomorphism, while maintaining the time complexity of the algorithm on which they were based.

Iso-canonical algorithms have many applications in the context of RDF, for example: the development of digital signatures for RDF graphs [12, 72, 62]; or to skolemize blank nodes consistently [18]; or for finding duplicate RDF graphs in large collections [34], such as the web. Fortunately, the worst cases of isomorphism do not occur frequently in published RDF graphs [34]. This induces two objectives for iso-canonical algorithms: on the one hand, to provide a correct result, even in the worst case; and to efficiently solve the graphs that are commonly encountered in practice, on the other hand. Examples of such algorithms are Nauty [55], Blabel [34], and URDNA2015 [74]. At the time of writing this document there is a working group in progress [49] to define, first, a standard to uniquely and deterministically calculate a hash of RDF Datasets, and second, to specify RDF Dataset Canonicalization algorithms. The Blabel and URDNA2015 algorithms are candidates to become a W3C standard to calculate the canonical version of an RDF graph modulo isomorphism. A base instance of a correct iso-canonical algorithm without optimizations is known as the κ-mapping [34]. In the following section, we address how the κ-mapping computes the canonical version of an RDF graph, and how to mechanically implement such an algorithm.

6.2 The κ-mapping algorithm

In the previous section we introduced the notions of canonical instances, as well as iso-canonical algorithms. We presented its theoretical status with respect to graphs, and how it has evolved and been applied with respect to RDF graphs. We have also presented some applications of iso-canonical algorithms, as well as the nascent interest of the W3C to specify and define a standard for iso-canonical algorithms. In this section we present how the κ-mapping algorithm operates, and we expose the main challenges for modeling iso-canonical algorithms in a dependent functional programming setting. Finally, we present a mechanized implementation of the κ-mapping algorithm.

The κ-mapping algorithm establishes the canonical graph in the following manner. The algorithm tries all possible combinations of mappings to relabel a graph G with the blank nodes of the set $\{b1, ..., b\kappa\}$, where κ is the number of blank nodes of G. Then, assuming a total order over the RDF graphs, κ-mapping chooses the minimum[1] graph from the ones produced after relabeling G under each of the κ mappings. In Figure 6.1 we illustrate two graphs: G and H, directed cycles of two blank nodes connected by the predicate `:p`, such that the blank nodes of G are `_:a` and `_:b`, and the ones from H are `_:x` and `_:y`. Regarding the graph G, there are two possible ways to map its blank nodes to $\{\texttt{_:0}, \texttt{_:1}\}$: κ_{G1} and κ_{G2}, which we define in the equations below. The same occurs for the graph H with two mappings: κ_{H1} and κ_{H2}, also defined below.

$$\kappa_{G1}(x) = \begin{cases} \texttt{_:0} & x \text{ is } \texttt{_:a} \\ \texttt{_:1} & x \text{ is } \texttt{_:b} \end{cases} \qquad \kappa_{H1}(x) = \begin{cases} \texttt{_:0} & x \text{ is } \texttt{_:x} \\ \texttt{_:1} & x \text{ is } \texttt{_:y} \end{cases}$$

$$\kappa_{G2}(x) = \begin{cases} \texttt{_:1} & x \text{ is } \texttt{_:a} \\ \texttt{_:0} & x \text{ is } \texttt{_:b} \end{cases} \qquad \kappa_{H2}(x) = \begin{cases} \texttt{_:1} & x \text{ is } \texttt{_:x} \\ \texttt{_:0} & x \text{ is } \texttt{_:y} \end{cases}$$

In this case, relabeling the graphs under the two mappings yields identical graphs, therefore both mapping minimize the candidate graphs for G and H. We illustrate the result of κ-mapping for the graphs G and H in Figure 6.2.

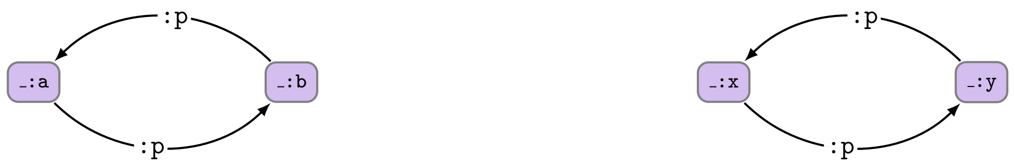


Figure 6.1: Illustration of the graphs G and H.

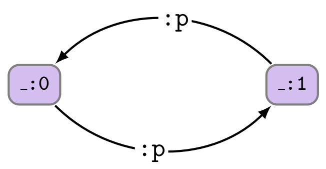


Figure 6.2: Illustration of the graphs (κ-mapping G) and (κ-mapping H).

[1]We explain the notion of order later in Subsection 6.2.1

The argument for certifying that κ-mapping is iso-canonical is based on its exhaustive search. The result is isomorphic to the input because each mapping establishes a bijection between the new set of blank nodes and the original one, that is, each possible mapping is a pre-isomorphism. Then, adjacency is preserved with respect to the input, because the result is definitionally the mapping of the pre-isomorphism onto the input graph. Given two isomorphic inputs, then these inputs give a set-equal result under κ-mapping. This is due to three observations, first, that the target set of blank nodes is independent of the input; second, that since inputs are isomorphic, each triple has the same IRIs and literals; and third, based on the first and the second argument, the resulting graph has identical triples. This last observation is due the fact that the κ-mapping that results in the minimal graph cannot map to different blank nodes. Finally, given two inputs that produce set-equal results under κ-mapping, the inputs are isomorphic by transitivity of isomorphism.

Before implementing κ-mapping in Coq, we first mechanically formalize the notion of an iso-canonical algorithm in Listing 35. The mechanized definition looks almost identical to Definition 6.1: given a function `M`, which takes as input an RDF graph parameterized by `I`, `B`, and `L`, and returns an RDF graph parameterized by the same types, it is an `isocanonical_mapping` if and only if, for any RDF graph `g`: `g` is isomorphic to `M g`, and, for all pair of graphs `g`, and `h`: applying `M` to `g` and `h` yields the same graph under set equality, if and only if, `g` and `h` are isomorphic. This is our target specification when defining κ-mapping in Coq.

```
Definition isocanonical_mapping (M : rdf_graph I B L -> rdf_graph I B L) :=
  forall g, iso g (M g) /\
    (forall g h, eqb_rdf (M g) (M h) <-> iso g h).
```

Listing 35: A mechanized definition of iso-canonical algorithms.

We remark on a few things regarding a mechanized definition of κ-mapping. Defining κ-mapping requires some aspects that we need to mechanize as well. We need, first, a total order relation for RDF graphs; second, we need to be able to generate an arbitrarily large target set of blank nodes in a deterministic manner; and third, we need a device keeping track of the canonical value assigned to each blank node.

6.2.1 Total order relation

Assuming a total order for the types `I`, `B` and `L`, we define an order for RDF terms, triples and sequence of triples. For terms, we implement an order `le_term` following the partial order defined by the SPARQL recommendation by the W3C [28]. Thus, the ascending order of the constructors declares the blank nodes to be the smaller kind of term, followed by the IRIs, and then by the literals. In case that terms coincide in their constructor, then we propagate the comparison to the corresponding parametric type's order. For triples we define `le_triple` by first comparing the subject, then the predicate, and finally the object. For ordering sequences of triples we define `le_st`, which defines the lexicographical order with respect to `le_triple`. Finally, applying `le_st` on sorted sequences corresponds to a relation between graphs, in which given two graphs G and H, if $G \subsetneq H$ then `le_st` G H, otherwise the smaller graph is the one containing the smallest triple not contained in the other graph. For each of the orders `le_*` we define:

- `lt_*` calculating less than.
- `meet_*` returning the minimum between the two inputs.
- `join_*` returning the maximum between the two inputs.
- Identities relating `lt_*` and `le_*`, `meet_*` and `lt_*`, and `join_*` and `lt_*`, such that for all `x` and `y`, it holds that:

  ```
  lt_* x y = (y != x) && (le_* x y) /\
  meet_* x y = (if lt_* x y then x else y) /\
  join_* x y = (if lt_* x y then y else x)
  ```
- And a proof that `le_*` is antisymmetric, transitive and total.

With these definitions we gain access to all the lemmas, notations and definitions from the math-comp's total order library. In particular, we gain access to the results about the maximum and the minimum.

6.2.2 Target set of blank nodes

For a given graph G parameterized by `I`, `B` and `L`, we choose the target set of blank nodes to be the first k natural numbers, meaning that the type of G's blank nodes would go from `B` to `nat`. However, we have defined the type of the mapping M in Listing 35 to preserve the type of blank nodes. Thus, we assume an injective function `nat_inj : nat -> B`, which injects natural numbers back to the type `B`. This function could be interpreted as a deserialization function for numbers.

6.2.3 Keeping track of blank node assignment

To establish an isomorphism between two graphs, we need to explicitly state the function linking them. Since iso-canonical algorithms produce results isomorphic to the input, we must find a way to trace the blank nodes that we will relabel, so we can then prove that adjacency is preserved. Also, since κ-mapping goes through all the possibilities, we have to trace the result for each function. To trace that every relabeling is an isomorphism, we design a structure `HashedData` based on the product types. The type `HashedData` is parameterized by two types `T` and `H`, representing a pair of the data-term `t : T`, paired with its current relabeling `h : H`. We illustrate its definition in Listing 36. The inductive type `hash` has a single constructor, composed of a pair: a data-term and its relabeling. The function `input` projects the data, while `current_hash` projects the current result to which an algorithm is relabeling the data. The function `mkHinput` allows for creating an inhabitant of the type `hash` by currying the argument that is a pair. In Listing 36 we provide two more definitions: `n0` as a default element, and `lookup_hash_default` to inspect the current hash of a given term, if it is a blank node.

```
Section HashedData.
  Variables (H T : Type).

  Inductive hash  := Hash of T * H.

  Definition input t :=  let: Hash th := t in th.1.

  Definition current_hash h := let: Hash th := h in th.2.

  Definition mkHinput (t : T) (h : H) := Hash (t, h).

End HashedData.
Definition n0 := 0.

Definition lookup_hash_default (hb : (term I (hash nat B) L) ) : nat :=
  if hb is Bnode hin then current_hash hin else n0.
```

Listing 36: Definition of `HashedData`.

6.3 A mechanized definition of κ-mapping

With these definitions we can now define a κ-mapping. We insist on making the definition with sequences of triples, and then extending it to RDF graphs. This makes it easier for us to carry out the subsequent proofs on the definition. Assuming we have three `orderTypes` : `I`, `B`, and `L`, we define κ-mapping progressively in Listing 37: given a sequence of triples `ts`, we compute all the permutations of its blank nodes (identified by `perms`). We then pair each permutation with the sequence of the first κ natural numbers as the target set of blank nodes, creating all possible assignments for the blank nodes (`perms_hash`). Next, we transform each pair in each permutation sequence into a `hash nat B`. Afterwards, we transform each of the permutations of the hash sequences into a function mapping the data to its current relabeling (`mus`). We explain how to construct such functions after finishing the explanation of the algorithm. We use each function in `mus` to relabel the sequence `ts` under them, obtaining all possible ways to label `ts` using the first κ natural numbers. All of these relabels are candidates for being the canonical instance of the graph (`cans`). Then, we sort every element in `cans` using `le_triple` as the order (`isocans`). Finally, we compute the maximum over the sequences in `isocans` using the empty sequence as the default term, which is the minimum sequence according to the order we defined. We make the observation that using the infimum as default is for a specific purpose: so that the only way to obtain the default as a result is if the input is also empty. Because of this, in order to avoid always returning the default value, we have to choose the maximum as canonical instead of the minimum. The maximum is computed using `join_st` according to our definitions.

To create the functions in `mus`, we develop `build_kmapping_from_seq`: a function constructor that generates a function dependent on a permutation of hashes. Given a sequence of terms `terms`, `build_kmapping_from_seq` returns a function that, for each input `b : B`, looks to see if `b` occurs as data in some hash in `terms`. In the case that this happens, it

```
Section kmapping.
  Variables I B L : orderType tt.

  Definition build_kmapping_from_seq
    (trms : seq (term I (hash nat B) L)) : B -> B :=
    fun b =>
    if has (eqb_b_hterm b) trms then
      let the_bnode :=
        nth (Bnode (mkHinput b n0)) trms (find (eqb_b_hterm b) trms) in
      nat_inj (lookup_hash_default the_bnode)
    else
      b.

  Definition k_mapping_ts (ts : seq (triple I B L)) : seq (triple I B L) :=
    let perms :=  permutations (get_bts ts) in
    let perms_hash := (map (fun s=> zip s (iota 0 (size s))) perms) in
    let all_maps := map (map Hash) perms_hash in
    let mus := map build_kmapping_from_seq all_maps in
    let cans := map (fun mu => (relabeling_seq_triple mu ts)) mus in
    let isocans := map (sort le_triple) cans
    in
    foldl join_st [::] isocans.

End kmapping.
```

Listing 37: Definition of κ-mapping in Coq.

returns the nth element and injects it to the type `B` using `nat_inj`, such that n meets the predicate of occurring as data. As a default element for `nth` we use the same blank node that was received as an argument, paired with `n0`. We remark that we will never fall into the default case of the `nth` function in `build_kmapping_from_seq`, because we make sure to use an index within the size of the list, which is ensured by the conditioning of the predicate `has (eqb_b_hterm b) terms` in the function `build_kmapping_from_seq`.

6.4 κ-mapping returns a well-formed graph

We defined κ-mapping in Coq for sequences of triples, but to extend the definition to apply for RDF graphs, we must prove that the result of κ-mapping is a well-formed graph. In other words, we must prove that for any graph `g`, it holds that `uniq (k_mapping_ts g)`. But before proving the lemma, we introduce a few notations to reduce the verbosity of the proofs. We denote by `hterm` the terms that have `hash nat B` as their blank nodes' type. We write `HBnode` to refer to the constructor that receives a pair `bn : (B * nat)` and returns a blank node of type `hterm`. A *hashed permutation of* `p` is the sequence formed by pairing the sequence `p` with the sequence of the first n natural numbers, and then mapping `HBnode`

```
Notation hn := (hash Order.NatOrder.orderType B).
Notation hterm := (term I hn L).

Definition HBnode p := @Bnode I hn L (mkHinput p.1 p.2).

Definition hash_kp p := [seq HBnode an | an <- zip p (iota 0 (size p))].

Definition build_map_k p := build_kmapping_from_seq (hash_kp p).
```

Listing 38: κ-mapping abstractions.

to the resulting sequence. For a permutation `[:: p1; ...; pn]` its hashed permutation is `[:: HBNode (p1,0); ... ; HBnode (pn,n)]`. Finally, we use `build_map_k p` to refer to the mapping resulting from applying `build_kmapping_from_seq` to the hashed perm of `p`. We mechanize these definitions in Listing 38.

We present the result to be proven in Listing 39. We notice that the series of consecutive `let-in` that we defined in `k_mapping_ts`, in Listing 37, disappears once we unfold the definition of `k_mapping_ts`. The challenge to prove this result is how to decompose the goal into smaller properties, tracking that the result is `uniq`. We approach the proof with a top-down strategy, that is, we assume something to prove that we can remove as a layer from the goal, and then we prove it, each time reducing the remaining goal to conclude. First we get rid of the layer that computes the maximum: the result of folding the maximum in a sequence can be the default element, or some element in the sequence. Our default term is trivially `uniq`; now it is left to prove that all the elements of the sequence are `uniq`. The sequence corresponds to the `let-in` identified by `isocans` in `k_mapping_ts`, that is, the relabeling of the input under each of the functions generated with `build_kmapping_from_seq`, which we then sort. As sorting does not interact with duplicates, it is enough to prove that every element in `cans` is `uniq`, which we prove because relabeling under a pre-isomorphism preserves well-formedness; then it suffices to prove that every function in `mus` is a pre-isomorphism. In other words, it suffices to show that the function constructor `build_kmapping_from_seq` builds pre-isomorphisms when its argument is a hash permutation. Since we want to prove that a certain mapping `mu` is a pre-isomorphism between the input and the relabeling of the input under `mu`, it is enough to prove that each function `mu` in `mus` is injective in the domain of the blank nodes of the input, where the proof of injectivity is based on the fact that both sequences paired with `zip` are duplicate-free, and therefore each hash is identified with a unique relabeling. Thus, we prove that κ-mapping preserves the well-formedness of an RDF graph. Such a proof, which we name as `uniq_k_mapping` in our project, allows us to define the κ-mapping algorithm for RDF graphs in Coq. We present its definition in Listing 40.

In the next two sections we present our strategies to prove κ-mapping is an iso-canonical algorithm, approaching first the proof stating that κ-mapping's result is isomorphic to the input, and then, that κ-mapping is a graph invariant whose result is a canonical instance of the input.

```
uniq
  (foldl join_st [::]
     [seq sort le_triple i
        | i <- [seq relabeling_seq_triple mu ts
           | mu <- [seq build_kmapping_from_seq i
              | i <- [seq [seq HBnode i | i <- i]
                 | i <- [seq zip s (iota 0 (size s))
                    | s <- permutations (get_bts ts)]]]]])
```

Listing 39: The goal to prove that `k_mapping_ts` returns a well-formed RDF graph.

```
Definition k_mapping (g : rdf_graph I B L) : rdf_graph I B L :=
  @mkRdfGraph I B L (k_mapping_ts (graph g)) (uniq_k_mapping g).
```

Listing 40: Definition of κ-mapping for RDF graphs.

6.5 κ-mapping returns a graph isomorphic to the input

In this section we cover the first goal to prove that κ-mapping is an iso-canonical algorithm. Formally we prove that for any graph G, G is isomorphic to the graph that results from applying κ-mapping to G, which in mechanized terms, corresponds to the following proposition: `forall g, iso g (k_mapping g)`. To achieve this, we use a few insights given by the proof of `uniq_k_mapping`. We reason again using a top-down strategy, but now we have many previous results. We start again by eliminating the layer computing the maximum. We cover the two cases; first, we prove that we can establish an isomorphism when fold returns the default value, and second, that we can also establish an isomorphism when the result of the fold is within the sequence that we fold. For the first case, due to the characteristics of our default value, it is enough to prove that if the result of κ-mapping is the default value, then the input is also the default value. Therefore, this case follows by the reflexivity of isomorphism. For the second case, by the properties of `map` and `sort`, we can show that there is a permutation of `g`'s blank nodes `p`, such that the result of κ-mapping is the relabeling of the input `build_map_k p`. Then, it suffices to show that `build_map_k p` is an isomorphism. We have already proven that for every permutation `q` of the blank nodes of `g`, `build_map_k q` is a pre-isomorphism; and that relabeling under it preserves the well-formedness of a graph, which leads us to conclude that it also preserves its adjacency. Therefore, the isomorphism for the second case is exactly `build_map_k p`. Thus, we have formalized a mechanized proof that the result of `k_mapping` is isomorphic to its input.

6.6 κ-mapping is an iso-canonical algorithm

In this section we prove the remaining goal of our application for `CoqRDF`, and conclude that κ-mapping is an iso-canonical algorithm. The remaining goal states that κ-mapping is a graph invariant, which returns the canonical instance of its input. In mechanized terms, we tackle the following formal statement:

```
forall g h, iso g h <-> eqb_rdf (k_mapping g) (k_mapping h).
```

(if) Given two RDF graphs `g` and `h`, which are isomorphic with `mu` as a witness. We approach it again with a top-down strategy removing the layer that computes the maximum; it is enough to prove that the sequence of sorted candidates of `g` has the same elements as the sequence of sorted candidates of `h`. Then, since `join_st` is an associative and idempotent operator, since `k_mapping` folds `join_st` using the neutral element of the operation, and since the folded sequences of sorted candidates have the same elements, the result is equal. Now we prove that the sequences of sorted candidates of `g` and `h` have the same elements. The proof uses of the direction of the isomorphism to prove both sequences have the same elements, thus, since isomorphism is symmetric, it suffices to show that given a member `sc` from `g`'s sequence of sorted candidates, then `sc` is also a member of `h`'s sorted candidates. By the properties of `map`, `sc` is the result of sorting a candidate `c` from the sequences of `g`'s candidates, which was built with a permutation of `g`'s blank nodes `p`, such that `c` is equal to relabeling `g` under `build_map_k p`. To prove that sorting the relabeling of `g` under `build_map_k p` is a member of `h`'s sorted candidates, it suffices to explicitly provide a candidate, which after sort is equal to `sc`, and a proof that this candidate is member of `h`'s candidates. We expose the relabeling of `h` under `build_map_k (map mu p)`, which is a member of the candidates of `h`, witnessed by the permutation `p` and the pre-isomorphism between `g` and `h`. Now we have to prove that sorting the relabeling of `g` under `build_map_k p` is equal to sorting the relabeling of `h` under `build_map_k (map mu p)`. We can remove the sorting layer showing that relabeling `g` under `build_map_k p`, and relabeling `h` under `build_map_k (map mu p)` are set-equal. Because of the isomorphism `mu` from `g` to `h`, for every triple `t` in `g`, `relabeling_triple mu t` is in `h`, which allows us to remove `h` from the goal by replacing the members from `h` with the members of the relabeling of `g` under `mu`. It remains to prove that relabeling `g` under `build_map p` is set-equal to relabeling `g` under `(build_map_k (map mu p)) \o mu`, for which it suffices to show that `build_map_k p` and `(build_map_k (map mu p)) \o mu` assign the same values for every blank node in the permutation `p`, which follows from the injectivity of `mu`, because it is a pre-isomorphism. Thus, we prove that for any pair of isomorphic graphs, they are set-equal under `k_mapping`.

(only if) Given two RDF graphs `g` and `h`, which are set-equal under `k_mapping`, `g` and `h` are isomorphic. Since `g` and `h` are set-equal under `k_mapping`, they are also isomorphic under `k_mapping` with the identity map as witness. Also, since for any graph, `k_mapping` returns a graph isomorphic to the input, then `g` is isomorphic to `k_mapping g`, idem for `h`. It follows that `g` and `h` are isomorphic by transitivity of isomorphism.

We have provided a proof that `k_mapping` returns a graph, which is isomorphic to the input; and that two graphs give a set-equal result under `k_mapping`, if and only if, the input graphs are isomorphic. We conclude that `k_mapping` is an iso-canonical algorithm with a fully mechanized proof. Finally, we generalize the application of `HashedData` which allows us to keep track of the isomorphism. This technique can be used for other algorithms to trace the mapping of a blank node throughout the computation. For this reason, we abstract the framework used for this purpose, and make it available in the file *Isocan.v* of our GitHub repository.

6.7 Summary

In this chapter we have presented the notions of canonical graph instances, and iso-canonical algorithms. We have introduced the theoretical notions regarding graphs, and their transfer to RDF; we have also motivated applications of iso-canonical algorithms in the context of RDF. Next, we presented a concrete instance of an iso-canonical algorithm: κ-mapping. We explained the challenges and mechanisms necessary to define κ-mapping in a functional programming paradigm. Afterwards, we mechanized a formal definition of iso-canonical algorithms, and developed a methodology that allows traceability of the relabeling of blank nodes. Subsequently, such a methodology allows us to prove, first, that κ-mapping returns well-formed graphs; second, that κ-mapping returns a graph isomorphic to its input. A third proof, stating that two graphs give a set-equal result under κ-mapping, if and only if, the input graphs are isomorphic. These two properties allows us to conclude that κ-mapping is an iso-canonical algorithm, with a fully mechanized proof. Finally, we abstract the framework that we have designed to trace the blank nodes, and make it available in our repository.

In the next part we conclude and discuss: the process of mechanizing the RDF model; the design choices posed by the formalization, which we then evaluate; the future work and perspectives for `CoqRDF`; and a summary of our contributions.

Part IV

Discussion and conclusion

Chapter 7

Discussion

In this chapter we discuss the formalization process and the design decisions we have made during development in Section 7.1. We evaluate the tools we used to implement `CoqRDF` and summarize the lessons we have learned in the process during Section 7.2.

7.1 On the design choices in the process

During the development of the thesis we made many design decisions. We name some of them, and the considerations we had to make them. Firstly, to model the RDF terms we had three possible representations: sets, types and finite types. Due to the definitional restrictions of the terms, types have turned out to be the sweet spot for modeling RDF terms, enforcing their disjunction, and defining their operations smoothly. Sets, on the one hand, would require us to define many external conditions to capture correct behaviors; and on the other hand, finite types induce the handling of low-level structures that are not relevant to the theory, and also worsening maintainability of the code.

Finding a comfortable structure for reasoning that enables the reusability of already formalized theories is not an easy task. Equivalence between representations does not guarantee equivalence in the work necessary to prove properties about them. We have had to solve this same dilemma when modeling RDF graphs where the problem of representing sets appears again, in this case, between duplicate-free sequences free and finite types, as we presented in Chapter 4. We mention that we also considered using the Doczkal and Pous [22] graph library, which formalizes several standard results from the graph theory literature, however we have discarded it for several reasons. The two most important are, first, that the representation of RDF graphs using directed graphs is good for gaining intuitions, but it would not be convenient for reasoning. Similar to `finTypes`, we would obtain a structure which would have to track more things than those we care about with no relief in the proof effort: the main complication comes from justifying the equality through a membership predicate, which would also be present in a representation using directed graphs. The second reason is that in order to inject a meta-theory developed in directed graphs back into RDF graphs, we would need to decode directed graphs back to RDF graphs, a procedure involving identifying cliques in directed graphs and mapping them to an RDF term. As far as we know, the clique theory provided by the graph library is not sufficient to decode graphs into RDF graphs.

Another case worth mentioning is that of isomorphism, and consequently also of pre-isomorphism. During development, we tried several representations: using bijections and functional extensionality; restricting the domain of bijections propositionally; and finally correlating two sequences through map. The latter is the one that gave us results. MathComp's philosophy of finding a decomposition of a proposition into decidable predicates systematically helped us to reduce the effort in proving properties of the proposition.

7.2 Evaluation

Through the development of `CoqRDF` I learned a lot of mathematics and concepts from the Semantic Web. Both communities have been very kind to me and for that I am very grateful; addressing the mechanization of a Semantic Web standard has been an entertaining and challenging work. Despite the learning curve of the use of proof assistants and their formalized libraries, their use is satisfactory. Math-comp provides a huge database for lemmas which we used intensively. The library of mathematical components suggests a methodology which I really enjoyed, having a good balance between abstraction and conciseness. By using it I have learned valuable lessons, I present a few of them below.

- To build up composable proofs with constructive arguments. This was an important lesson when tackling goals with many layers, as any proof involving the κ-mapping algorithm.
- The value of proving with pen and paper before a mechanized attempt of the formalization. A lot of time can be wasted by trying to mechanize results that are not true, thinking that the reason lies in the representation.
- Many of the software design principles are applicable in mathematics. Refactoring what we had already mechanized has constantly driven our progress, either by generalizing lemmas, or by finding abstractions that we have not seen in the first instance.
- Intuitions and proofs in pen and paper have guided us to mechanize our library. However, we were surprised that most of the proof effort was in the details. Justifications that fundamentally seem obvious usually are not.

We have presented some decisions we have made throughout this mechanization, as well as an appraisal of our tools. With this we move on to the next chapter, where we present our contributions, future work and perspectives, and our conclusions.

Chapter 8

Conclusion

In this chapter we summarize our contributions in Section 8.1, and present future work and perspectives for `CoqRDF` in Section 8.2.

8.1 Contributions

To the best of our knowledge, we have developed the first mechanized library to reason about RDF graphs: `CoqRDF`. We used the Coq proof assistant and the library of mathematical components using the ssreflect methodology. We provide modules to reason about RDF terms, triples and RDF graphs. Each module provides operations for reasoning about RDF operations at the level of its different components. We define operations mainly for reasoning about the relabeling of blank nodes, and the projections of the terms and blank nodes of an RDF graph. Subsequently, we mechanically define the notion of isomorphism using `CoqRDF`. We provide mechanized results about RDF isomorphism, for example that it is an equivalence relation. On top of RDF isomorphism we mechanize the definition of iso-canonical algorithms. Then, using `CoqRDF` we implement the κ-mapping algorithm, and develop a methodology to trace the mapping of a blank node throughout computation. This methodology allows us to prove that κ-mapping returns well-formed graphs; that κ-mapping returns graphs that are isomorphic to the input; and that two graphs give a set-equal result under κ-mapping, if and only if, the input graphs are isomorphic. These properties mechanically verify that κ-mapping is an iso-canonical algorithm. Finally, we modularized the methodology to trace blank nodes, and make it available as a library allowing the methodology to be applied to other mechanized algorithms in the future. The development we have presented is available at https://github.com/Tvallejos/CoqRDF.

We address our perspectives for `CoqRDF` in the following section.

8.2 Future work and perspectives

A mechanized proof that κ-mapping is an iso-canonical algorithm directly enables the mechanization of other iso-canonical algorithms, such as the candidates for the RDF canonization standard. Other directions in the line of RDF isomorphism include: to explore the terrain of graph pattern matching, and thereby mechanize other semantic web standards, such as SHACL [40], which define how to validate that RDF graphs satisfy sets of structural conditions. Another alternative would be the mechanization of compact structures on RDF, such as the Ring [2], and enabling reasoning about RDF under different representations; an approach which can be performed using univalent parametricity [67, 68], making available the transferring of definitions and theorems between equivalent representations of a model. Another possible path is to study the data models and semantics of RDF [18], or to mechanize other standards built on top of RDF, such as its query language, SPARQL [28]. We believe that we have contributed with a first stone towards a mechanized toolchain of web standards based on RDF.

Bibliography

[1] Andrew W. Appel. *Verified Functional Algorithms*, volume 3 of *Software Foundations*. Electronic textbook, 2023. Version 1.5.4, http://softwarefoundations.cis.upenn.edu.

[2] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-Case Optimal Graph Joins in Almost No Space. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 102–114, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457256. URL `https://doi.org/10.1145/3448016.3457256`.

[3] László Babai. Graph Isomorphism in Quasipolynomial Time [Extended Abstract]. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, page 684–697, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341325. doi: 10.1145/2897518.2897542. URL `https://doi.org/10.1145/2897518.2897542`.

[4] László Babai. Canonical Form for Graphs in Quasipolynomial Time: Preliminary Report. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, page 1237–1246, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367059. doi: 10.1145/3313276.3316356. URL `https://doi-org.uchile.idm.oclc.org/10.1145/3313276.3316356`.

[5] Sean Bechhofer and Andreas Harth. The Semantic Web Challenge 2014. *Journal of Web Semantics*, 35:141–141, 12 2015. doi: 10.1016/j.websem.2015.11.001.

[6] Dave Beckett. RDF/xml syntax specification (revised). W3C recommendation, W3C, February 2004. https://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[7] Véronique Benzaken and Evelyne Contejean. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 249–261. ACM, 2019. doi: 10.1145/3293880.3294107. URL `https://doi.org/10.1145/3293880.3294107`.

[8] Véronique Benzaken, Évelyne Contejean, Chantal Keller, and Eunice Martins. A Coq formalisation of SQL's execution engines. In *ITP 2018 - International Conference on Interactive Theorem Proving*, volume 10895 of *Lecture Notes in Computer Science*, pages 88–107, Oxford, United Kingdom, July 2018. Springer. doi: 10.1007/978-3-319-94821-8_6. URL `https://hal.science/hal-01716048`.

[9] Véronique Benzaken, Sarah Cohen-Boulakia, Evelyne Contejean, Chantal Keller, and Rébecca Zucchini. A Coq formalization of data provenance. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 152–162. ACM, 2021. doi: 10.1145/3437992.3439920. URL `https://doi.org/10.1145/3437992.3439920`.

[10] Gavin Carothers and Eric Prud'hommeaux. RDF 1.1 turtle. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-turtle-20140225/.

[11] Jeremy Carroll and Graham Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C recommendation, W3C, February 2004. https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[12] Jeremy J. Carroll. Signing RDF Graphs. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003, Proceedings*, volume 2870 of *Lecture Notes in Computer Science*, pages 369–384. Springer, 2003. doi: 10.1007/978-3-540-39718-2_24. URL `https://doi.org/10.1007/978-3-540-39718-2_24`.

[13] Boutheina Chetali and Quang-Huy Nguyen. Industrial Use of Formal Methods for a High-Level Security Evaluation. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods*, pages 198–213, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-68237-0.

[14] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 510–524. ACM, 2017. doi: 10.1145/3062341.3062348. URL `https://doi.org/10.1145/3062341.3062348`.

[15] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.*, 11(11):1482–1495, 2018. doi: 10.14778/3236187.3236200. URL `http://www.vldb.org/pvldb/vol11/p1482-chu.pdf`.

[16] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Inf. Comput.*, 76 (2/3):95–120, 1988. doi: 10.1016/0890-5401(88)90005-3. URL `https://doi.org/10.1016/0890-5401(88)90005-3`.

[17] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. ISBN 978-3-540-46963-6.

[18] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[19] Richard Dapoigny and Patrick Barlatier. Modeling Ontological Structures with Type Classes in Coq. In Heather D. Pfeiffer, Dmitry I. Ignatov, Jonas Poelmans, and Nagarjuna Gadiraju, editors, *Conceptual Structures for STEM Research and Education, 20th International Conference on Conceptual Structures, ICCS 2013, Mumbai, India, January 10-12, 2013. Proceedings*, volume 7735 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2013. doi: 10.1007/978-3-642-35786-2_11. URL `https://doi.org/10.1007/978-3-642-35786-2_11`.

[20] Tomás Díaz, Federico Olmedo, and Éric Tanter. A Mechanized Formalization of GraphQL. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 201–214, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370974. doi: 10.1145/3372885.3373822. URL `https://doi.org/10.1145/3372885.3373822`.

[21] Reinhard Diestel. *Graph Theory*. Springer Publishing Company, Incorporated, 5th edition, 2017. ISBN 3662536218.

[22] Christian Doczkal and Damien Pous. Graph Theory in Coq: Minors, Treewidth, and Isomorphisms. *J. Autom. Reason.*, 64(5):795–825, 2020. doi: 10.1007/s10817-020-09543-2. URL `https://doi.org/10.1007/s10817-020-09543-2`.

[23] Renee Dopplick. Open source Coq wins ACM Software System Award, 2014. URL `https://techpolicy.acm.org/2014/04/open-source-coq-wins-acm-software-system-award/`.

[24] Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. doi: 10.1007/978-3-540-87827-8_28. URL `https://doi.org/10.1007/978-3-540-87827-8_28`.

[25] 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 Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013. doi: 10.1007/978-3-642-39634-2_14. URL `https://doi.org/10.1007/978-3-642-39634-2_14`.

[26] 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, 2016. URL `https://inria.hal.science/inria-00258384`.

[27] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2nd edition, 2016. ISBN 1107150302.

[28] Steven Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[29] John Harrison, Josef Urban, and Freek Wiedijk. History of Interactive Theorem Proving. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. North-Holland, 2014. doi: https://doi.org/10.1016/B978-0-444-51624-4.50004-6. URL `https://www.sciencedirect.com/science/article/pii/B9780444516244500046`.

[30] Ivan Heman, Ben Adida, Manu Sporny, and Mark Birbeck. RDFa 1.1 primer. Technical report, W3C, August 2013. https://www.w3.org/TR/2013/NOTE-rdfa-primer-20130822/.

[31] Ivan Herman and Pierre-Antoine Champin. RDF Dataset Canonicalization and Hash Working Group Charter, 2020. URL `https://w3c.github.io/rch-wg-charter/`.

[32] José-Miguel Herrera, Aidan Hogan, and Tobias Käfer. BTC-2019: The 2019 Billion Triple Challenge Dataset. In *The Semantic Web – ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II*, page 163–180, Berlin, Heidelberg, 2019. Springer-Verlag. ISBN 978-3-030-30795-0. doi: 10.1007/978-3-030-30796-7_11. URL `https://doi.org/10.1007/978-3-030-30796-7_11`.

[33] Jose Miguel Herrera, Aidan Hogan, and Tobias Käfer. Billion Triple Challenge (BTC) 2019 Dataset, April 2019.

[34] Aidan Hogan. Canonical Forms for Isomorphic and Equivalent RDF Graphs: Algorithms for Leaning and Labelling Blank Nodes. *ACM Trans. Web*, 11(4):22:1–22:62, 2017. doi: 10.1145/3068333. URL `https://doi.org/10.1145/3068333`.

[35] Aidan Hogan. *The Web of Data.* Springer, 2020. ISBN 978-3-030-51579-9. doi: 10.1007/978-3-030-51580-5. URL `https://doi.org/10.1007/978-3-030-51580-5`.

[36] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. Everything you always wanted to know about blank nodes. *J. Web Semant.*, 27-28:42–69, 2014. doi: 10.1016/j.websem.2014.06.004. URL `https://doi.org/10.1016/j.websem.2014.06.004`.

[37] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. *Knowledge Graphs.* Number 22 in Synthesis Lectures on Data, Semantics, and Knowledge. Springer, 2021. ISBN 9783031007903. doi: 10.2200/S01125ED1V01Y202109DSK022. URL `https://kgbook.org/`.

[38] William Howard. The formulae-as-types notion of construction. page 479–490, 1980.

[39] Inria, 2021. URL `https://compcert.org/`.

[40] Dimitris Kontokostas and Holger Knublauch. Shapes constraint language (SHACL). W3C recommendation, W3C, July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[41] Zhangsheng Lai, Aik Beng Ng, Liang Ze Wong, Simon See, and Shaowei Lin. Dependently Typed Knowledge Graphs. *CoRR*, abs/2003.03785, 2020. URL `https://arxiv.org/abs/2003.03785`.

[42] Markus Lanthaler and Gregg Kellogg. RDF 1.1 test cases. W3C note, W3C, February 2014. https://www.w3.org/TR/2014/NOTE-rdf11-testcases-20140225/.

[43] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax. Technical report, W3C, October 1997. https://www.w3.org/TR/WD-rdf-syntax-971002/.

[44] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C recommendation, W3C, February 1999. https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[45] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7): 107–115, jul 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL `https://doi.org/10.1145/1538788.1538814`.

[46] Pierre Letouzey. *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq*. Theses, Université Paris Sud - Paris XI, July 2004. URL `https://theses.hal.science/tel-00150912`.

[47] Pierre Letouzey. Extraction in Coq: An Overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008. doi: 10.1007/978-3-540-69407-6_39. URL `https://doi.org/10.1007/978-3-540-69407-6_39`.

[48] Haixia Li and Li Yan. A Temporal RDF Model for Multi-grained Time Information Modeling. In *DSIT 2021: 4th International Conference on Data Science and Information Technology, Shanghai, China, July 23 - 25, 2021*, pages 9–14. ACM, 2021. doi: 10.1145/3478905.3478908. URL `https://doi.org/10.1145/3478905.3478908`.

[49] Dave Longley, Gregg Kellogg, and Dan Yamamoto. RDF dataset canonicalization. W3C working draft, W3C, Oct 2023. https://www.w3.org/TR/rdf-canon/.

[50] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer. doi: 10.1007/978-3-642-39634-2_5. URL `https://inria.hal.science/hal-00816703`.

[51] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, September 2022. doi: 10.5281/zenodo.7118596. URL `https://doi.org/10.5281/zenodo.7118596`.

[52] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On Blank Nodes. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC*

2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I, volume 7031 of *Lecture Notes in Computer Science*, pages 421–437. Springer, 2011. doi: 10.1007/978-3-642-25073-6_27. URL https://doi.org/10.1007/978-3-642-25073-6_27.

[53] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984. ISBN 978-88-7088-228-5.

[54] Rudolf Mathon. A Note on the Graph Isomorphism counting Problem. *Inf. Process. Lett.*, 8(3):131–132, 1979. doi: 10.1016/0020-0190(79)90004-8. URL https://doi.org/10.1016/0020-0190(79)90004-8.

[55] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014. ISSN 0747-7171. doi: https://doi.org/10.1016/j.jsc.2013.09.003. URL https://www.sciencedirect.com/science/article/pii/S0747717113001193.

[56] Peter Mika and Jim Hendler. The Semantic Web challenge, 2008. *J. Web Semant.*, 7 (4):271, 2009. doi: 10.1016/S1570-8268(09)00065-1. URL https://doi.org/10.1016/S1570-8268(09)00065-1.

[57] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

[58] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015. URL https://inria.hal.science/hal-01094195.

[59] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

[60] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018. Version 5.5. http://www.cis.upenn.edu/ bcpierce/sf.

[61] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018. Version 5.5. http://www.cis.upenn.edu/ bcpierce/sf.

[62] Craig Sayers and Alan Karp. Computing the digest of an RDF graph. Technical report, HP, 05 2004. https://www.hpl.hp.com/techreports/2003/HPL-2003-235R1.pdf.

[63] Andy Seaborne and Gavin Carothers. RDF 1.1 n-triples. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[64] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371076. URL `https://doi.org/10.1145/3371076`.

[65] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *J. Autom. Reason.*, 64(5):947–999, 2020. doi: 10.1007/s10817-019-09540-0. URL `https://doi.org/10.1007/s10817-019-09540-0`.

[66] Manu Sporny, Gregg Kellogg, and Markus Lanthaler. JSON-ld 1.0. W3C recommendation, W3C, January 2014. https://www.w3.org/TR/2014/REC-json-ld-20140116/.

[67] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *Proc. ACM Program. Lang.*, 2(ICFP):92:1–92:29, 2018. doi: 10.1145/3236787. URL `https://doi.org/10.1145/3236787`.

[68] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. The Marriage of Univalence and Parametricity. *J. ACM*, 68(1):5:1–5:44, 2021. doi: 10.1145/3429979. URL `https://doi.org/10.1145/3429979`.

[69] Q-Success team. Usage statistics of structured data formats for websites, 2009. URL `https://w3techs.com/technologies/overview/structured_data`.

[70] The Coq Development Team. The Coq Proof Assistant, June 2023. URL `https://doi.org/10.5281/zenodo.8161141`.

[71] Dominik Tomaszuk and David Hyland-Wood. RDF 1.1: Knowledge representation and data integration language for the web. *Symmetry*, 12(1):84, 2020. doi: 10.3390/sym12010084. URL `https://doi.org/10.3390/sym12010084`.

[72] Giovanni Tummarello, Christian Morbidoni, Paolo Puliti, and Francesco Piazza. Signing individual fragments of an RDF graph. In Allan Ellis and Tatsuya Hagino, editors, *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005 - Special interest tracks and posters*, pages 1020–1021. ACM, 2005. doi: 10.1145/1062745.1062848. URL `https://doi.org/10.1145/1062745.1062848`.

[73] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015. doi: 10.1145/2699407. URL `https://doi.org/10.1145/2699407`.

[74] Dan Yamamoto, Gregg Kellogg, and Dave Longley. RDF Dataset Canonicalization. W3C working draft, W3C, October 2023. https://www.w3.org/TR/2023/WD-rdf-canon-20231004/.