Bachelor's Thesis Computing Science

Formalizing the query language of Semantic MediaWiki

Marijn van Wezel
s1040392

June 27, 2023

First supervisor/assessor:
dr. Robbert Krebbers

Second assessor:
dr. Patrick van Bommel

Radboud University

Acknowledgements

I am extremely grateful to dr. Robbert Krebbers for his time and energy spent helping me write this thesis. Without his feedback and the weekly meetings, this thesis would not exist. I especially want to thank him for answering my many questions about Coq, and for his help on structuring the text.

I also want to thank my friends and family for their support and their time spent proofreading this thesis.

Abstract

Semantic MediaWiki is a popular extension to the well-known wiki-engine MediaWiki, used to power Wikipedia, that allows editors to annotate pages with structured data and link pages together. In this thesis, we formalize a fragment of the query language of Semantic MediaWiki. To make our semantics simpler and more principled, we only present a direct set-based semantics for a minimal core language. We also provide a type-sound translation from the fragment that we formalize to this core language. We provide the formalizations both in written mathematics, as well as the Coq proof assistant. Finally, we use the Coq mechanized semantics to prove a number of semantic equivalences for queries in the core language.

Contents

Chapter 1

Introduction

Semantic MediaWiki is a popular extension to the well-known wiki-engine MediaWiki, used to power Wikipedia. The first stable version was released in 2007 (Semantic MediaWiki contributors, 2020c), and it is now used on numerous websites, such as Liquipedia[1], TranslateWiki[2] and FSF's Free Software Directory[3]. A conference for users and developers of Semantic MediaWiki is also held every year[4].

Semantic MediaWiki enables editors of a wiki to annotate pages with structured data. This data can be queried and shown on other pages, used to create lists or be used for searching. Data can also be exported for external analysis. To query this structured data, Semantic MediaWiki comes with a domain-specific query language. Because everyone is able to contribute to a wiki, regardless of their technical knowledge, the main focus of this query language is simplicity. This simplicity makes queries easy to read and write, but also leads to ambiguity and invalid assumptions. For example, consider the query:

```
[[Has zip code::1337]] AND [[Has zip code::!1337]]
```

You might expect this query to never return any results, but the law of non-contradiction does not apply. In fact, the above query returns any city that has the zip-code 1337 and at least one other zip-code. Also consider the following, very similar query:

```
[[Has zip code::1337]] OR [[Has zip code::!1337]]
```

Instead of returning every page, this query returns only pages that have a zip-code.

[1]https://liquipedia.net
[2]https://translatewiki.net
[3]https://directory.fsf.org
[4]https://www.semantic-mediawiki.org/wiki/SMWCon

It is important that the behaviour of the query language is well-understood by editors, so that they can make sure that the queries they write give the results they desire. An often used approach to reason about a query or programming language is by defining a formal semantics for it. This allows you to *mathematically* reason about the meaning and behaviour of a language, enabling you to do proofs of correctness, equivalence, termination, *et cetera* (Floyd, 1967). To make proofs easier and more reliable, a *mechanized* semantics, i.e. a semantics that has been defined in a proof assistant, such as Coq (Coq Development Team, 2021), HOL4 (Slind & Norrish, 2008) or Isabelle (Wenzel et al., 2008), is most ideal.

Mechanized semantics have already been defined for many programming and query languages, and they have been used to study the mathematical meaning of programs (Appel and Blazy, 2007; Krebbers, 2015; Chu et al., 2017), create a formally-verified compiler for C (Leroy, 2009) and SQL (Benzaken et al., 2022) and give a machine-checked safety proof for a (subset) of Rust's type system (Jung et al., 2018), among other things. Defining a (mechanized) semantics for Semantic MediaWiki's query language will be the main focus of this thesis.

Bao et al., 2008 are as far as we know the only authors to have worked on formalizing Semantic MediaWiki. They give (indirect) semantics by translating queries to logic programs. We will give a direct set-based semantics instead, using well-defined set operations, such as union and filter, since it is easier to reason with and to mechanize. We mechanize our semantics in the Coq proof assistant. The mechanization in Coq not only makes it easier and more reliable to prove properties about queries, it also makes it possible to relate our work to the work mentioned above in the future. For instance, to create a verified compiler for Semantic MediaWiki queries.

The application of our semantics is twofold: (i) It serves as utterly precise documentation of the query language. This is not only useful for users of Semantic MediaWiki, but also for developers of Semantic MediaWiki that want to make sure that their code does not contain breaking changes. (ii) It serves as a basis for proofs about the query language.

Our semantics are based on a core language, called **Core Semantic MediaWiki**. Semantic MediaWiki queries contain quite a bit of superfluous syntax and duplicate constructs, which would make the semantics unnecessarily complicated. Instead, we define our semantics by first elaborating (translating) from **Surface Semantic MediaWiki**, a language that closely resembles a subset of the query language of Semantic MediaWiki, to **Core Semantic MediaWiki**, for which we give the semantics directly. We have proven this elaboration to be type-sound in Coq. The different parts of the semantics are graphically displayed in Figure 1.1.

We have determined the semantics by studying the reference manual (Semantic MediaWiki contributors, 2019), by looking at compiled queries and by looking at the results of queries. Furthermore, we have proven some properties about the semantics in Coq to gain more confidence that it indeed constitutes the semantics of Semantic MediaWiki.

The Coq formalization and the installation instructions can be found on Zenodo (van Wezel, 2023).

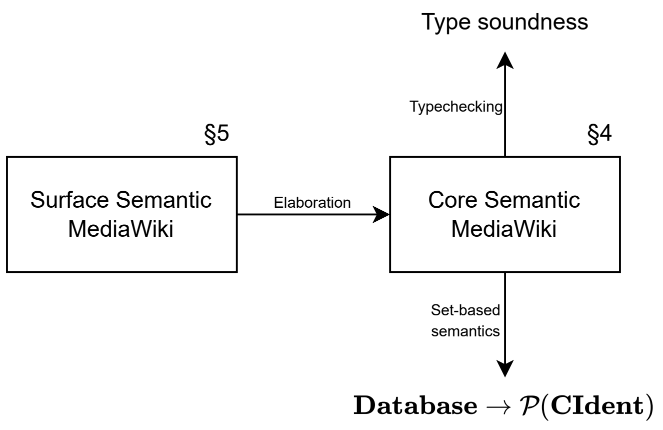

Figure 1.1: High-level overview of the semantics.

In this thesis, we make a number of assumptions and simplifications, which are listed below.

- We have not formalized the *printing language* of Semantic MediaWiki.

  The printing language is responsible for taking the result of a query and presenting it to the user as a table, calendar, diagram, *etc*.

- We take the database as a given and therefore do not formalize how data is modified.

- We do not formalize any inferencing that Semantic MediaWiki does (Semantic MediaWiki contributors, 2020a), nor do we formalize synonyms.

  Semantic MediaWiki automatically does simple inferencing for sub-categories and sub-properties. For example, if a page is in the category ‘A’, and that category is a sub-category of ‘B’, then a query to ‘B’ will still return the page. It is also possible to add synonyms for pages by using redirects: if a page ‘A’ redirects to a page ‘B’, then replacing ‘B’ in any query with ‘A’ will give the same result.

- We only formalize a fragment of the query language. A complete list of features that we formalize is given in Chapter 5. A list of notable missing features is given in Chapter 8.

- We do not formalize a parser for Semantic MediaWiki queries.
- Queries are compiled to a back-end (SQL, Elasticsearch or SPARQL) by Semantic MediaWiki. We do not formalize this compilation process.
- We take the SQL-implementation of Semantic MediaWiki as the reference implementation.

  Semantic MediaWiki supports a number of different back-ends, such as SQL or Elasticsearch. When the result of a query is different between back-ends, and the documentation does not resolve this conflict, we use the behaviour of the SQL-implementation to define the semantics. An example of such a conflict is the behaviour of the inequality comparator on (namespaced) identifiers, as explained in Section 5.3.1.

Contributions. The main contributions of this thesis are:

- A formal semantics for a fragment of the Semantic MediaWiki query language, both in written mathematics as well as in the Coq proof assistant. These semantics serve both as documentation of the query language, as well as a basis for proofs.
- A number of semantic equality proofs about **Core Semantic MediaWiki** queries in Coq, as well as a proof of the type-soundness of the elaboration.

Outline. Chapter 2 introduces the query language of Semantic MediaWiki through a number of examples. Chapter 3 introduces special notation that is used throughout this thesis. In Chapter 4 we introduce Core Semantic MediaWiki, and give its formal semantics. Chapter 5 formalizes the translation from Surface Semantic MediaWiki to Core Semantic MediaWiki. In Chapter 6, we give a short explanation of our Coq implementation. Finally, in Chapter 7 we discuss related work and Chapter 8 concludes this thesis and gives possible directions for future work.

Chapter 2

Semantic MediaWiki by example

2.1 Introduction

In this chapter, we will cover the basics of Semantic MediaWiki and informally introduce the concepts that we will formalize in Chapter 4 and Chapter 5.

This chapter is divided into the following sections:

(i) In Section 2.2 we will briefly look at what Semantic MediaWiki is.

(ii) In Section 2.3 we look at how to organize a Semantic MediaWiki wiki and we introduce the database that we will use for examples throughout this thesis.

(iii) In Section 2.4 we will give concrete examples for all features that we will formalize in this thesis.

2.2 Semantic MediaWiki

Semantic MediaWiki [1] is an extension to the wiki-engine MediaWiki [2] — best known for powering Wikipedia [3] — that enables users to store and query information on a wiki. It comes with a domain-specific query language, which is what we will formalize in this thesis. It also has a secondary language that is used to display, order and work with results, but this language is out-of-scope for this thesis.

[1]https://www.semantic-mediawiki.org

[2]https://www.mediawiki.org

[3]https://www.wikipedia.org

Because Semantic MediaWiki is an *extension* to MediaWiki, it is good to take a moment and clarify which things are part of MediaWiki and which things are added by Semantic MediaWiki.

A MediaWiki installation consists of pages uniquely identified by their title, for instance `Amsterdam` might be a page that talks about Amsterdam. These pages are organized into namespaces by prefixing the title with '`<namespace>:`'. For example, the page `Talk:Amsterdam` lives in the `Talk` namespace. If a title is not prefixed, it lives in the `main` namespace (for example, our imaginary page about `Amsterdam` lives there). We will call a namespaced title an *identifier* and a non-namespaced title an *article name*.

Pages can be categorized using categories. Unlike namespaces, a page can be in any (finite) number of categories, including no categories at all. Furthermore, categories can contain pages from different namespaces. For example, the page about `Amsterdam` might be in the `Capital` and the `City` category.

Semantic MediaWiki extends MediaWiki to add the concept of *properties*, which allow you to add structured data to pages and link pages together. Our recurring example `Amsterdam` may for instance be linked to the page `The Netherlands` using the property `Is located in`. This link creates a so-called *semantic triple* (Lassila & Swick, 1999), which can be read like a sentence:

> `Amsterdam` *is located in* `The Netherlands`.

The concept of semantic triples comes from the Semantic Web [4], of which Semantic MediaWiki is an implementation.

Properties in Semantic MediaWiki have a global datatype. By default, this datatype is `page`, which is the datatype used to link pages together. However, other data types, such as `string` or `number`, are also possible and allow you to store structured data directly on a page, which can then be queried and shown on other pages.

2.3 Topography wiki

Suppose you are a topographer and you want to publish some data that you have collected. After looking at the available options, you have decided to use Semantic MediaWiki for this. This section will explain how you can best organize such a wiki, and introduce the database that will be used for examples throughout the rest of this thesis.

The topography wiki will contain pages about a number of topography concepts, namely cities, countries and continents. You organize the wiki as a hierarchy of topographical areas. You do this by linking pages using the (used-defined)

[4]https://www.w3.org/standards/semanticweb/

	Is located in	Has zip code	Has population	Is capital	*Categories*
Amsterdam	The Netherlands	1023;1024;1025	821,752	True	City; Capital
Barcelona	Spain	08001;08002;08003	1,620,000	False	City
Berlin	Germany	10115;10117;10119	3,645,000	True	City; Capital
Cairo	Egypt	3753450;3755220	9,540,000	True	City; Capital
Nijmegen	The Netherlands	3769;5211	170,681	False	City
Egypt	Africa		109,300,000		Country
Germany	Europe		83,200,000		Country
The Netherlands	Europe		17,530,000		Country
Spain	Europe		47,420,000		Country
Africa			1,216,000,000		Continent
Europe			746,400,000		Continent

Table 2.1: The topography wiki.

`Is located in` property. More specifically, you link cities to countries and countries to continents using this property. For example, the page `Amsterdam` would be linked to `The Netherlands`, which would be linked to `Europe`. Note that Semantic MediaWiki does not place any constraints on what pages can be linked, and it is up to the wiki's administrator to organize their wiki in such a way that it forms a logical hierarchy.

You categorize your topography wiki by adding the following categories:

- `Capital`, containing all capital cities;
- `City`, containing all cities;
- `Country`, containing all countries;
- `Continent`, containing all continents.

And you also create the following (additional) properties:

- `Has zip code` (`string`), containing the zip code(s) for a city;
- `Has population` (`number`), containing the number of inhabitants of an area;
- `Is capital` (`boolean`), containing whether the city is a capital.

In Semantic MediaWiki, all properties are a set. It is up to the wiki administrator to only assign a single value if that is appropriate (for example, multiple values for `Has population` would be nonsense).

We define an example database in Table 2.1. Note that the data is only an example and it may not be accurate or complete. Multiple values for the same property are separated using a semicolon. The categories for the page are given in the italicized column *Categories*.

2.4 Features

In this section, we will look at the different features of Semantic MediaWiki that we will formalize, and informally describe their behaviour. We will also give a number of examples.

2.4.1 Identifier selectors

Semantic MediaWiki allows you to query pages based on their title. We call these selectors ‘identifier selectors’. The simplest identifier selector selects a single page. For example, the query `[[Amsterdam]]` would only select the page `Amsterdam`.

It is possible to use comparators in an identifier selector, such as `!` to select all identifiers except one, or > to select all identifiers (lexicographically) greater or equal. For example, to get all pages that start with the letter `A` or `B`, we can do:

```
[[>A]] [[<<C]]
```

Where > means greater than *or equal*, << means strictly less than, and the juxtaposition of the selectors a conjunction (intersection of the results of both selectors). This gives `Amsterdam`, `Barcelona` and `Berlin`.

Finally, it is also possible to select pages based on whether they live in a specific namespace:

```
[[Talk:+]]
```

This gives the empty set, since our wiki does not have any pages in the `Talk` namespace.

2.4.2 Category selectors

As mentioned in Section 2.2, categories can be used to group pages of similar subjects together. For example, the category `City` would contain all cities, and the category `Country` would contain all countries.

Semantic MediaWiki allows you to query these categories like so:

```
[[Category:City]]
```

Which would return all cities (`Amsterdam`, `Barcelona`, `Berlin`, `Cairo` and `Nijmegen`).

2.4.3 Property selectors

Semantic MediaWiki is also capable of selecting pages based on the properties that are on a page. For example, we could write

```
[[Is located in::The Netherlands]]
```

to get all pages for which the property ‘Is located in’ has the value ‘The Netherlands’, which in our case would give `Amsterdam` and `Nijmegen`.

Similar to identifier selectors is it possible to use comparators in a property selector. The available comparators are:

Syntax	Name	Description
`!`	inequality	The property must have a value that is unequal.
`>`	greater than or equal	The property must have a value that is greater or equal.
`>>`	greater than	The property must have a value that is strictly greater.
`<`	less than or equal	The property must have a value that is less or equal.
`<<`	less than	The property must have a value that is strictly less.

Note that the `!` comparator does not select pages that do not have the property at all, instead, it only selects pages that have a value unequal to the value specified. This can lead to unintuitive behaviour when a property has multiple values. Consider the following query:

```
[[Category:City]] [[Has zip code::!1023]]
```

Instead of selecting all cities that do not have the zip code `1023`, it selects all cities that (also) have a zip code unequal to `1023`. In particular, any page without the property `Has zip code` will not be a result of this query, and any city with multiple ZIP-codes will always be a result of this query. The result of the above query is therefore `Amsterdam`, `Barcelona`, `Berlin`, `Cairo` and `Nijmegen`.

A special case of the property selector is the wildcard selector, which selects all pages that have at least one value for a particular property. The syntax for the wildcard selector is very similar to that of a regular property selector, but instead of writing an comparator and a value, you simply write a `+`. For example, to select all pages with the property `Has population`, you can write `[[Has population::+]]`.

Finally, it is also possible to select pages based on multiple different comparators and values by putting `||` between them. For example, to select all capital cities where the population is more than five million *or* less than one million, you can use the query:

```
[[Category:Capital]] [[Has population::<<1000000||>>5000000]]
```

Which would yield `Amsterdam` and `Cairo`.

2.4.4 Subqueries and property-chains

Subqueries allow you to select pages based on the results of another query. This can be useful if you have a large number of values you want to select on or if the values you are selecting on are dynamic. Consider the following query:

```
[[Category:City]] [[Is located in::Germany||Spain||...]]
```

This query selects all cities located in a country in Europe. Since there are many countries in the Europe, and countries can split or merge at any time, enumerating all of them is not feasible. Luckily, Semantic MediaWiki thought of this and has support for subqueries. We can use the following query to first generate a list of all countries in Europe:

```
[[Is located in::Europe]]
```

And then we can use the results of that query in our original query. Instead of copying the results and pasting them in our query, we can use a subquery:

```
[[Category:City]]
[[Is located in::<q>[[Is located in::Europe]]</q>]]
```

This pattern is so common that Semantic MediaWiki provides syntactic sugar, called *property chains*, for it. The following query will give identical results to the query above:

```
[[Category:City]] [[Is located in.Is located in::Europe]]
```

Subqueries (and by extension property chains) can be of arbitrary depth.

2.4.5 Disjunctions and conjunctions

Semantic MediaWiki also supports disjunctions (*or*'s) and conjunctions (*and*'s) between selectors. To create a disjunction, you can use the keyword `OR`:

```
[[Category:Country]] OR [[Category:City]]
```

To create a conjunction, you can use juxtaposition or the keyword `AND`:

```
[[Category:Country]] [[Is located in::Europe]]
```

or

```
[[Category:Country]] AND [[Is located in::Europe]]
```

Without explicit groupings, queries are in disjunctive normal form, i.e. a disjunction of conjunctions. For example, the query `A OR B AND C OR D` should be read as `A OR (B AND C) OR D`. Instead of the more conventional opening and closing parentheses, Semantic MediaWiki uses <q> and </q> respectively for groupings. For example:

```
[[Category:Capital]] <q>[[Berlin]] OR [[Nijmegen]]</q>
```

Chapter 3

Notation

In this chapter, we define some notation that will be used throughout the rest of this thesis.

3.1 Record types

We occasionally use record types instead of product types (tuples). Record types are similar to product types, as they are also used to hold multiple values of possibly distinct types. However, unlike product types, they automatically introduce a named projection to read each value. For example, consider the following type:

$$\{\mathsf{head} : \mathbb{Z},\ \mathsf{tail} : \mathbb{R}\}$$

Instead of using anonymous projections, we can use the functions head and tail to denote the values in the tuple. That is, for any value $c = (z, r) \in \{\mathsf{head} : \mathbb{Z},\ \mathsf{tail} : \mathbb{R}\}$, we can write head c to denote the first value (z) and tail c to denote the second value (r).

3.2 Multi-argument function types

For functions that accept multiple input arguments, we use currying. For brevity, we shall omit parentheses in function types. Specifically, for any type A, B and C, we shall write the type of a function as:

$$A \rightarrow B \rightarrow C$$

to denote

$$A \rightarrow (B \rightarrow C)$$

3.3 Partial functions

Partial functions are occasionally in this thesis when no reasonable result exists for some input of a function. For any type A and B, we shall write the type of a partial function f as:

$$A \rightharpoonup B$$

Where $\rightharpoonup$ denotes the partial relation. This makes it possible to combine partial functions with total functions. For example, we can write $A \rightarrow B \rightharpoonup C$ to denote a *total* function that yields the partial function $B \rightharpoonup C$ when the first argument is applied.

We write $\perp$ to indicate that an expression is undefined. For example, we can write $f\ a := \perp$ to denote that f is undefined for any input $a \in A$.

Furthermore, we shall write $f\ a \downarrow$ for the predicate that f is defined for input $a \in A$ and $f\ a \uparrow$ for the predicate that f is not defined for input $a \in A$.

Finally, we define the convention that undefined values propagate. That is, if the result of a sub-expression is undefined, then the entire expression is also undefined.

Chapter 4

Core Semantic MediaWiki

4.1 Introduction

In this chapter, we will describe Core Semantic MediaWiki. Core Semantic MediaWiki closely resembles regular Semantic MediaWiki, but omits duplicate or superfluous syntax. For example, the different disjunctions allowed by Semantic MediaWiki are generalized into a single construct (see Example 4.2.1) and all values are explicitly typed. These simplifications make the semantics easier to grasp and more principled. The expressiveness of Semantic MediaWiki queries is retained through a translation from our formalized fragment of Semantic MediaWiki (**Surface Semantic MediaWiki**) to Core Semantic MediaWiki queries (see Chapter 5).

We start by giving the abstract syntax of Core Semantic MediaWiki (Section 4.2), then we will describe the data model (Section 4.3) that we use. Next, in Section 4.4, we will explain how datatypes work. Finally, we will give the semantics of Core Semantic MediaWiki (Section 4.5).

4.2 Abstract syntax

The first step in defining the semantics for any language is specifying its syntax (Floyd, 1967). In this thesis, we will use Backus-Naur form (BNF) to describe the syntax. It is important to note that we only define the *abstract syntax*. The abstract syntax captures the structure of a query, but omits details that enable unique parse trees to be constructed. It is the job of the *concrete syntax* to provide enough details to be able to construct unique parse trees. When we talk about *syntax*, we mean the *abstract syntax*, unless otherwise noted.

We start by listing the various meta-variables that will be used throughout this section and the rest of this chapter to describe the different syntactic categories

(Nielson & Nielson, 1992). For Core Semantic MediaWiki queries, we use the convention that:

- x will range over names, **Name**;
- i will range over identifiers, **CIdent**;
- v will range over values, **CVal**;
- ω will range over comparators, **CComp**;
- s will range over selectors, **CSel**;
- q will range over queries, **CQuery**.

These meta-variables may be primed or subscribed. So, q_1, q_2 and q' all stand for constructs in the category **CQuery**.

The first syntactic category, **Name**, contains all names that MediaWiki allows. To avoid conflicts with other parts of the syntax, the rules for which names are allowed are complex (MediaWiki contributors, 2022). Since names do not influence the semantics, we will not formalize the syntax of **Name** in this thesis and define it to be a string.

Definition 4.2.1. We define the abstract syntax of Core Semantic MediaWiki below. Terminals are given in the `typewriter` font, whereas non-terminals are given in *italics*.

$x \in$ **Name**	:=	string	(name)
$i \in$ **CIdent**	::=	x`:`x	(identifier)
$v \in$ **CVal**	::=	`true` | `false`	(boolean)
		| numeral	(number)
		| string	(string)
$\omega \in$ **CComp**	::=	`=` | `!=` | `>` | `<`	(comparator)
$s \in$ **CSel**	::=	`Category:`x	(category selector)
		| x`:+`	(namespace selector)
		| x`::+`	(existence selector)
		| x`::<q>` q `</q>`	(subquery selector)
		| ω x	(article name selector)
		| x`::`ω v	(value selector)
$q \in$ **CQuery**	::=	`[[`s`]]`	(selector)
		| q_1 `OR` q_2	(disjunction)
		| q_1 `AND` q_2	(conjunction)

For any identifier $i = x_1 : x_2 \in \mathbf{CIdent}$, we shall write namespace i to refer to the namespace of i (x_1), and article i to refer to the article name of i (x_2).

Example 4.2.1. In Surface Semantic MediaWiki, a disjunction can be written two forms:

(i) `[[Has zip code::1023||10115]]`

(ii) `[[Has zip code::1023]] OR [[Has zip code::10115]]`

Both queries are identical. Since (i) is just syntactic sugar for (ii), Core Semantic MediaWiki only supports the de-sugared form (ii).

4.3 Database function

The evaluation of any query language will depend on the contents of the database. A semantic function for a query language will therefore have to take an argument that determines the values in the database. In this section, we will formalize this argument in the form of a *database function*.

Recall from the preliminaries (Section 2.2) that a MediaWiki installation consists of a finite number of pages. For Semantic MediaWiki, the relevant information for each page is limited to:

(i) The categories on that page,

(ii) The values for each property on that page, and,

(iii) The identifier (title) of that page.

In reality, the database contains much more information about pages (such as old revisions, edit summaries, etc.), but this is not relevant for the semantics, and therefore not included in our formalization of the database.

We can easily capture (i) through a set. Since categories are simply pages that live in the 'Category' namespace, we can identify a category solely by its name. Therefore, the categories for any particular page is simply a constant of the type:

$$\mathcal{P}(\mathbf{Name})$$

Where $\mathcal{P}$ refers to the power set.

We will capture (ii) by using a function going from the name of a property to the set of values for that property. Similar to categories, properties are also pages (but in the 'Property' namespace) and can therefore also be identified by a name in **Name**. Since a property can also contain references to other pages (when the type of the property is page), the values of any property are actually a set containing both members of **CVal** and members of **CIdent**. Therefore, the function capturing the properties on a page will be of the type:

$$\textbf{Name} \rightarrow \mathcal{P}(\textbf{CIdent} + \textbf{CVal})$$

If a property does not exist on a page, this function will return the empty set. Note that this function will only return an inhabited set for a finite number of input values, as the database can only be finitely large.

Definition 4.3.1. We shall define the type **Page** as a record that contains both the categories of a page and the values of each property of a page.

$$\begin{aligned}\textbf{Page} := \{ & \\ & \text{cats} : \mathcal{P}(\textbf{Name}), \\ & \text{props} : \textbf{Name} \rightarrow \mathcal{P}(\textbf{CVal} + \textbf{CIdent}) \\ \} & \end{aligned}$$

Finally, we capture (iii) by using a (partial) map going from an identifier to a page.

Definition 4.3.2. The database function is a *partial* function going from an identifier to a page. That is, the database function is an element of the type:

$$\textbf{Database} := \textbf{CIdent} \rightharpoonup \textbf{Page}$$

The result of a function in **Database** is $\perp$ if the identifier is unknown.

For any database $D \in \textbf{Database}$, we write $\text{dom}\ D$ to refer to the domain of D, that is, the identifiers known to Semantic MediaWiki.

Note that the database function will only be defined for finitely many input values, as a database can only be finitely large. $\text{dom}\ D$ will therefore always be a finite set.

4.4 Datatypes

Every property in Semantic MediaWiki has a global datatype set by an editor of the wiki. This datatype changes how values are compared, parsed and stored in the database. While in Surface Semantic MediaWiki (Chapter 5), every value is a plain string, in Core Semantic MediaWiki, a value must have an explicit type. Instead of more conventional languages that parse values based on differences in syntax (quotes always mean a string, numbers always mean an integer, etc.), Semantic MediaWiki parses values differently based on this (silent) datatype. All values in the database for a property must have the correct type. For example, if a property has the global datatype `number`, all values in the database for that property must also be numbers.

Consider the boolean property `Is capital` and the string property `Has name`. In the query `[[Is capital::No]]`, the value `No` would be elaborated to the boolean value `false`, while in the query `[[Has name::No]]` [1], `No` would be parsed as the string `No` (assuming that the datatype of `Has name` is `string`).

To capture the concept of datatypes, we define a total function going from the name of a property to a type.

Definition 4.4.1. We inductively define the datatypes as:

$$\tau \in \textbf{Datatype} \quad ::= \quad \texttt{boolean} \mid \texttt{string} \mid \texttt{number} \mid \texttt{page}$$

Definition 4.4.2. We define a *typing* function, which is a total function going from a property name to a datatype. That is, it has the type:

$$\textbf{Typing} := \textbf{Name} \to \textbf{Datatype}$$

We assume that every property is explicitly assigned a type. In reality, the type of a property is often omitted and defaults to `page`.

Definition 4.4.3. We say that a Core Semantic MediaWiki query is well-typed if and only if:

- All property selectors are well-typed, and
- All subquery selectors are well-typed.

A property selector is well-typed if and only if:

- Its property has type `boolean` and its value is `true` or `false`, or,
- Its property has type `string` and its value is a *string*, or,
- Its property has type `number` and its value is a *number*.

Any other property selectors are not well-typed. A subquery selector is well-typed if and only if its property has type `page` and the subquery is well-typed.

4.5 Semantics

In this section, we will give the formal semantics of Core Semantic MediaWiki.

[1] `No` is a small village located in Denmark.

4.5.1 Semantics of comparators

In this subsection, we define the semantics of comparators.

The (ordinal) comparison operations ($>$ and $<$) are dependent on the types of their operands, and we assume that the semantics of these operations are well-defined for all relevant types.

Definition 4.5.1 (Article name comparators)**.** We define the semantic function C_a to compare article names as:

$$C_\mathsf{a} \ : \ \mathbf{CComp} \to \mathbf{Name} \to \mathbf{Name} \to \mathbb{B}$$

$$\begin{aligned} C_\mathsf{a}[\![\mathtt{=}]\!]\ x_1\ x_2 &:= x_1 = x_2 \\ C_\mathsf{a}[\![\mathtt{!=}]\!]\ x_1\ x_2 &:= x_1 \neq x_2 \\ C_\mathsf{a}[\![\mathtt{>}]\!]\ x_1\ x_2 &:= x_1 > x_2 \\ C_\mathsf{a}[\![\mathtt{<}]\!]\ x_1\ x_2 &:= x_1 < x_2 \end{aligned}$$

In Semantic MediaWiki, article names are compared lexicographically.

Definition 4.5.2 (Value comparators)**.** We define the partial semantic function C_v to compare values as:

$$C_\mathsf{v} \ : \ \mathbf{CComp} \to \mathbf{CVal} \to \mathbf{CVal} \rightharpoonup \mathbb{B}$$

$$\begin{aligned} C_\mathsf{v}[\![\mathtt{=}]\!]\ v_1\ v_2 &:= v_1 = v_2 \\ C_\mathsf{v}[\![\mathtt{!=}]\!]\ v_1\ v_2 &:= v_1 \neq v_2 \\ C_\mathsf{v}[\![\mathtt{>}]\!]\ v_1\ v_2 &:= v_1 > v_2 \\ C_\mathsf{v}[\![\mathtt{<}]\!]\ v_1\ v_2 &:= v_1 < v_2 \end{aligned}$$

C_v is only well-defined for values of the same datatype and will be $\perp$ for non-matching datatypes.

4.5.2 Semantics of queries and selectors

In this subsection, we will define the semantics of the queries and selectors in Core Semantic MediaWiki. The result of executing a query in Semantic MediaWiki is a *set* of identifiers, and not a more complicated value such as a table, as is the case for SQL (Benzaken et al., 2022). This makes the semantics set-based, and we can use well-defined set operations such as union ($\cup$), intersection ($\cap$) and set-builder to define it.

The semantics of the main query connectives `OR` and `AND` can be modelled exactly as the set operations union ($\cup$) and intersection ($\cap$) respectively. We capture this in the function Q (Definition 4.5.3) of the type:

$$Q \ : \ \mathbf{CQuery} \to \mathbf{Database} \to \mathcal{P}(\mathbf{CIdent})$$

The selectors can be modelled as a filter on the set of pages that are known to Semantic MediaWiki. For this, we use set-builder notation. The predicate for this set-builder will be a function that takes the syntax of the selector, the identifier of the page in question and the database as arguments and returns whether the identifier matches the selector (see Example 4.5.1). We capture this through the function $\mathcal{S}$ of the type:

$$\mathcal{S} \;:\; \textbf{CSel} \rightarrow \textbf{CIdent} \rightarrow \textbf{Database} \rightarrow \mathbb{B}$$

Example 4.5.1. Consider the query with only a single selector:

```
[[Category:City]]
```

To answer this query, we can look at each page in the database and check whether it indeed contains the category `City`. If we filter (remove) all pages that do not contain the category `City`, we'll be left with the set of pages that satisfy the above query.

$\mathcal{S}$ and $\mathcal{Q}$ will be mutually recursive because of subqueries. A query can contain a selector that can contain a query. For the semantics of Core Semantic MediaWiki, we assume that the query is well-typed.

Remark. The semantic brackets $[\![\cdot]\!]$ have nothing to do with the brackets around a selector (`[[ · ]]`), even though they look similar.

Definition 4.5.3. We mutually recursively define the semantic functions $\mathcal{S}$ and $\mathcal{Q}$ as:

$$\begin{aligned}
\mathcal{S}[\![\texttt{Category:}x]\!]\; i\; D &:= x \in \mathsf{cats}\; (D\; i) && \text{(category)}\\
\mathcal{S}[\![x\texttt{:+}]\!]\; i\; D &:= \mathsf{namespace}\; i = x && \text{(namespace)}\\
\mathcal{S}[\![x\texttt{::+}]\!]\; i\; D &:= \mathsf{props}\; (D\; i)\; x \neq \emptyset && \text{(property existence)}\\
\mathcal{S}[\![x\texttt{::<q>}\; q\; \texttt{</q>}]\!]\; i\; D &:= \mathsf{props}\; (D\; i)\; x \cap \mathcal{Q}[\![q]\!]\; D \neq \emptyset && \text{(subquery)}\\
\mathcal{S}[\![\omega\; x]\!]\; i\; D &:= \mathcal{C}_{\mathsf{a}}[\![\omega]\!]\; (\mathsf{article}\; i)\; x && \text{(article name)}\\
\mathcal{S}[\![x\texttt{::}\omega\; v']\!]\; i\; D &:= \exists\, v \in \mathsf{props}\; (D\; i)\; x.\; \mathcal{C}_{\mathsf{v}}[\![\omega]\!]\; v\; v' && \text{(property value)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{Q}[\![q_1\; \texttt{AND}\; q_2]\!]\; D &:= \mathcal{Q}[\![q_1]\!]\; D \cap \mathcal{Q}[\![q_2]\!]\; D && \text{(conjunction)}\\
\mathcal{Q}[\![q_1\; \texttt{OR}\; q_2]\!]\; D &:= \mathcal{Q}[\![q_1]\!]\; D \cup \mathcal{Q}[\![q_2]\!]\; D && \text{(disjunction)}\\
\mathcal{Q}[\![\texttt{[[}\, s\, \texttt{]]}]\!]\; D &:= \left\{\; i \in \mathsf{dom}\; D \;\middle|\; \mathcal{S}[\![s]\!]\; i\; D \;\right\} && \text{(selector)}
\end{aligned}$$

4.6 Explanation of the semantics

In this section, we will give some additional intuition behind the more difficult parts of the semantics to hopefully make them clearer.

The semantic function for selectors, $\mathcal{S}$, yields a proposition on whether the given identifier matches the selector for the given database. For most selectors, this proposition is relatively straightforward. We shall only take a deeper look at subquery and property value selectors, as those are the least intuitive.

4.6.1 Property value selectors

Because properties can contain multiple values, Semantic MediaWiki selects a page if there is at least one value that satisfies the comparator. To capture this, the semantics uses an existential quantification that checks if there is a value v in the database that satisfies the comparator with the value in the selector, v'. This definition leads to somewhat unintuitive behaviour for the not-equals comparator (`!=`). Instead of only selecting pages that do not have the specified value, it selects all pages that have at least one value that is unequal to the specified value. Therefore, any page with multiple values for that property will always match. Consider the query:

```
[[Has zip code::!=1054]]
```

You might expect `Amsterdam` to not be included in this query (`1054` is a zip code for Amsterdam), but that is not the case! Since Amsterdam has many zip codes unequal to `1054`, the exists quantifier is satisfied and `Amsterdam` will be included in the results.

Because we assume that queries are well-typed, we know that v and v' have the same datatype, and we can therefore be sure that the comparison is well-defined. As such, we do not check this explicitly.

4.6.2 Subquery selectors

Before we go into the formal semantics of subqueries, lets take a brief moment to look at how subqueries should *intuitively* behave. We do this by looking at a concrete example. Consider the query:

```
[[Is located in::<q>[[Is located in::Europe]]</q>]]
```

To execute this query, Semantic MediaWiki must check for every page if the property `Is located in` shares at least one value with the results of the query `[[Is located in::Europe]]`.

In the semantics, this is captured using an intersection. The left-hand side of the intersection, props $(D\ i)\ x$, is the values in the database for the property on which the subquery is executed, and the right-hand side, $\mathcal{Q}[\![q]\!]\ D$, is the result of the subquery. A page is included in the results if and only if this intersection is not empty, i.e. the result of the subquery and the property share at least one value.

4.7 Equivalence proofs

Using our mechanized semantics (see Chapter 6), we have proven a number of semantic equivalences for Core Semantic MediaWiki queries. In this section, we will only give a table of the proven equivalences, and the actual proofs are given only in the Coq development, which can be found on Zenodo (van Wezel, 2023).

We have proven that the following queries are semantically equivalent:

- `[[`x`::<q>`q_1` OR `q_2`</q>]]` = `[[`x`::<q>`q_1`</q>]] OR [[`x`::<q>`q_2`</q>]]`
- `[[`x`::+]]` = `[[`x`::=`v`]] OR [[`x`::!=`v`]]`
- `[[`x_1`:+]]` = `[[`x_1`:+]] AND ([[=`x_2`]] OR [[!=`x_2`]])`
- `[[`x_1`:+]]` = `[[`x_1`:+]] AND ([[=`x_2`]] OR [[>`x_2`]] OR [[<`x_2`]])`
- `[[=`x_2`]] OR [[!=`x_2`]]` = `[[=`x_2`]] OR [[>`x_2`]] OR [[<`x_2`]]`
- q_1 `AND` q_2 = q_2 `AND` q_1
- q_1 `OR` q_2 = q_2 `OR` q_1

Chapter 5

Surface Semantic MediaWiki

5.1 Introduction

In this chapter, we will describe Surface Semantic MediaWiki, which is the fragment of the Semantic MediaWiki query language that we will formalize in this thesis. It contains most of the important features of the Semantic MediaWiki query language, namely:

- Category selectors
- Identifier selectors
- Property selectors
- Subqueries and property-chains
- Disjunctions and conjunctions
- Comparators on identifiers and property values
- The `page`, `number`, `boolean` and `string` datatypes

We start by giving the abstract syntax of Surface Semantic MediaWiki in Section 5.2. The semantics of Surface Semantic MediaWiki are not given directly. Instead, a translation from Surface Semantic MediaWiki to a simpler language, Core Semantic MediaWiki, is given in Section 5.3. The semantics of Core Semantic MediaWiki are given in Chapter 4.

5.2 Abstract syntax

Similar to what we did for Core Semantic MediaWiki, we will first list the different *syntactic categories* and associated meta-variables. These meta-variables will be used to range over constructs (members) of the respective syntactic category (Nielson & Nielson, 1992).

Definition 5.2.1 (Syntactic categories)**.** We define the following syntactic categories for Surface Semantic MediaWiki:

- x will range over names, **Name**;
- v will range over values, **SValue**;
- ω will range over comparators, **SComp**;
- r will range over references (identifiers and article names), **SRef**;
- p will range over properties, **SProp**;
- c will range over category selectors, **SCSel**;
- ι will range over identifier selectors, **SISel**;
- ϕ will range over property selectors, **SPSel**;
- q will range over queries, **SQuery**.

Identically to Core Semantic MediaWiki, the first syntactic category, **Name**, contains all names that MediaWiki allows. Since names do not influence the behaviour of queries in any way, we will not formalize their syntax in this thesis and define it to be a string.

The syntactic category **SValue** captures the different data types that Semantic MediaWiki has. We define it to be a string, which gets elaborated differently based on the type of the associated property (see Section 4.4).

Definition 5.2.2 (Abstract syntax)**.** We mutually inductively define the abstract syntax of Surface Semantic MediaWiki as:

$$
\begin{array}{rcll}
x \in \textbf{Name} & := & \text{string} & \text{(string)} \\
v \in \textbf{SValue} & := & \text{string} & \text{(value)} \\
\omega \in \textbf{SComp} & ::= & \texttt{=} \mid \texttt{!=} & \text{(equality and inequality)} \\
 & & \mid \texttt{>} \mid \texttt{<} & \text{(greater and less than)} \\
 & & \mid \texttt{>=} \mid \texttt{<=} & \text{(greater and less than or equal)} \\
r \in \textbf{SRef} & ::= & x \texttt{:} x & \text{(identifier)} \\
 & & \mid x & \text{(article name)} \\
p \in \textbf{SProp} & ::= & x & \text{(single property)} \\
 & & \mid x \,\texttt{.}\, p & \text{(property chain)} \\
c \in \textbf{SCSel} & ::= & x & \text{(singleton category)} \\
 & & \mid c_1 \texttt{ || } c_2 & \text{(disjunction)} \\
\iota \in \textbf{SISel} & ::= & x \texttt{:+} & \text{(namespace selector)} \\
 & & \mid \omega\, r & \text{(reference selector)} \\
 & & \mid \iota_1 \texttt{ || } \iota_2 & \text{(disjunction)} \\
\phi \in \textbf{SPSel} & ::= & \texttt{+} & \text{(property existence)} \\
 & & \mid \omega\, v & \text{(value selector)} \\
 & & \mid \texttt{<q>}\, q \,\texttt{</q>} & \text{(subquery)} \\
 & & \mid \phi_1 \texttt{ || } \phi_2 & \text{(disjunction)} \\
q \in \textbf{SQuery} & ::= & \texttt{[[Category:}c\texttt{]]} & \text{(category selector)} \\
 & & \mid \texttt{[[}\iota\texttt{]]} & \text{(identifier selector)} \\
 & & \mid \texttt{[[}p\texttt{::}\phi\texttt{]]} & \text{(property selector)} \\
 & & \mid q_1 \texttt{ OR } q_2 & \text{(disjunction)} \\
 & & \mid q_1 \texttt{ AND } q_2 & \text{(conjunction)}
\end{array}
$$

5.3 Elaboration

In this section, we will describe the translation (elaboration) from Surface Semantic MediaWiki to Core Semantic MediaWiki. Given some typing environment in **Typing** (see Section 4.4), the elaboration function $\mathcal{E}$ (see Definition 5.3.4) turns a Surface Semantic MediaWiki query into a Core Semantic MediaWiki query.

The result of the elaboration may be undefined if the input query is not well-typed. This happens when a value does not correspond to its associated datatype. For example, a query like `[[Has population::true]]` would not be well-typed, since `true` is not a number. If the result of the elaboration is well-defined, then the resulting core query will be well-typed. We have proven this property in Coq (see Section 5.3.5).

To make the elaboration slightly more comprehensible, we will split it up into the following functions:

- $\mathcal{E}_{\text{ident}}$: **SISel** → **CQuery** (Definition 5.3.1)
- $\mathcal{E}_{\text{cat}}$: **SCSel** → **CQuery** (Definition 5.3.2)
- $\mathcal{E}_{\text{val}}$: **Name** → **SComp** → **SValue** → **Datatype** ⇀ **CQuery** (Definition 5.3.3)
- $\mathcal{E}_{\text{prop}}$: **SProp** → **SPSel** → **Typing** ⇀ **CQuery** (Definition 5.3.4)
- $\mathcal{E}$: **SQuery** → **Typing** ⇀ **CQuery** (Definition 5.3.4)

We will assume that a function val : **SValue** → **Datatype** ⇀ **CValue** that parses a value exists. It is a partial function, because the result will be undefined if the value is invalid for the given type. We do not define these functions explicitly, because their exact implementation is not relevant for the behaviour of queries.

5.3.1 Elaboration of identifier selectors

In this subsection, we give the elaboration rules of identifier selectors.

Definition 5.3.1. We recursively define the function $\mathcal{E}_{\text{ident}}$ to elaborate a surface identifier selector (**SISel**) to a core query.

$$\begin{aligned}
\mathcal{E}_{\text{ident}}[\![x:+]\!] &:= \texttt{[[}\,x:+\,\texttt{]]}\\
\mathcal{E}_{\text{ident}}[\![=x]\!] &:= \texttt{[[}\,:+\,\texttt{]] AND [[}=x\,\texttt{]]}\\
\mathcal{E}_{\text{ident}}[\![>=x]\!] &:= \texttt{[[}>x\,\texttt{]] OR [[}=x\,\texttt{]]}\\
\mathcal{E}_{\text{ident}}[\![<=x]\!] &:= \texttt{[[}<x\,\texttt{]] OR [[}=x\,\texttt{]]}\\
\mathcal{E}_{\text{ident}}[\![\omega\, x]\!] &:= \texttt{[[}\,\omega\, x\,\texttt{]]}\\
\mathcal{E}_{\text{ident}}[\![\,!=x_1:x_2]\!] &:= \texttt{[[}\,!=x_2\,\texttt{]]}\\
\mathcal{E}_{\text{ident}}[\![>=x_1:x_2]\!] &:= \texttt{[[}\,x_1:+\,\texttt{]] AND ([[}>x_2\,\texttt{]] OR [[}=x_2\,\texttt{]])}\\
\mathcal{E}_{\text{ident}}[\![<=x_1:x_2]\!] &:= \texttt{[[}\,x_1:+\,\texttt{]] AND ([[}<x_2\,\texttt{]] OR [[}=x_2\,\texttt{]])}\\
\mathcal{E}_{\text{ident}}[\![\omega\, x_1:x_2]\!] &:= \texttt{[[}\,x_1:+\,\texttt{]] AND [[}\,\omega\, x_2\,\texttt{]]}\\
\mathcal{E}_{\text{ident}}[\![\iota_1 \texttt{ || } \iota_2]\!] &:= \mathcal{E}_{\text{ident}}[\![\iota_1]\!] \texttt{ OR } \mathcal{E}_{\text{ident}}[\![\iota_2]\!]
\end{aligned}$$

The reason that = x elaborates to a conjunction between a namespace selector and an article name selector, is because the article name x is implicitly assumed to be in the *main* namespace. This is not the case for other comparators (see Example 5.3.1). The elaboration rule for != x_1: x_2 ignores the namespace completely in the SQL-implementation, but doesn't ignore it in the Elasticsearch implementation. Since there is no documentation on this, we use the current behaviour of the SQL-implementation for the elaboration.[1]

Example 5.3.1. Consider the queries `[[>Berlin]]` and `[[=Berlin]]`. Although these queries look very similar, they should be interpreted quite differently. The first query, `[[>Berlin]]`, selects all pages of which the *article name* is (lexicographically) greater than `Berlin`, regardless of which namespace the page is in. However, the second query will only select exactly the page `Berlin` in the *main* namespace.

5.3.2 Elaboration of categories

In this subsection, we give the elaboration rules of category selectors.

Definition 5.3.2. We define the function $\mathcal{E}_{\mathsf{cat}}$ to elaborate a surface category selector **(SCSel)** to a core query.

$$\begin{aligned}\mathcal{E}_{\mathsf{cat}}\,[\![x]\!] &:= \texttt{[[}\,\texttt{Category:}x\,\texttt{]]}\\ \mathcal{E}_{\mathsf{cat}}\,[\![c_1 \texttt{ || } c_2]\!] &:= \mathcal{E}_{\mathsf{cat}}[\![c_1]\!]\ \texttt{OR}\ \mathcal{E}_{\mathsf{cat}}[\![c_2]\!]\end{aligned}$$

Example 5.3.2. The query `[[Category:City||Country]]` would be elaborated to:

`[[Category:City]] OR [[Category:Country]]`

5.3.3 Elaboration of values

In this subsection, we give the elaboration rules for values.

Definition 5.3.3. We define the function $\mathcal{E}_{\mathsf{val}}$ to elaborate a surface value selector to a core query.

$$\begin{aligned}\mathcal{E}_{\mathsf{val}}[\![x\texttt{::>=}\,v]\!]\ \tau &:= \mathcal{E}_{\mathsf{val}}[\![x\texttt{::>}v]\!]\ \tau\ \texttt{OR}\ \mathcal{E}_{\mathsf{val}}[\![x\texttt{::=}v]\!]\ \tau\\ \mathcal{E}_{\mathsf{val}}[\![x\texttt{::<=}\,v]\!]\ \tau &:= \mathcal{E}_{\mathsf{val}}[\![x\texttt{::<}v]\!]\ \tau\ \texttt{OR}\ \mathcal{E}_{\mathsf{val}}[\![x\texttt{::=}v]\!]\ \tau\\ \mathcal{E}_{\mathsf{val}}[\![x\texttt{::}\omega\ v]\!]\ \tau &:= \texttt{[[}\,x\texttt{::}\omega\ (\mathsf{val}\ v\ \tau)\,\texttt{]]}\end{aligned}$$

Recall that the val function parses a value differently based on the type of the value, and may be undefined if v is not valid for τ. In a query such as `[[Is capital::True]]`, `True` would be elaborated to a `boolean`, while in a query such as `[[Has zip code::True]]`, it would be elaborated to a `string`.

[1]We have reported this as a bug on GitHub (SemanticMediaWiki/SemanticMediaWiki#5482).

5.3.4 Elaboration of property selectors and queries

In this subsection, we give the mutually recursive elaboration rules of property selectors and queries. The definition is mutually recursive because of subqueries.

Definition 5.3.4. We define the mutually recursive functions $\mathcal{E}$ and $\mathcal{E}_{\mathsf{prop}}$ to elaborate a surface query and a surface property selector respectively to a core query.

$$
\begin{aligned}
\mathcal{E}_{\mathsf{prop}}[\![x\texttt{::}\omega\ v]\!]\ \Gamma &:= \begin{cases} \texttt{[[}\,x\texttt{::<q>}\ \mathcal{E}_{\mathsf{ident}}[\![\omega\ v]\!]\ \texttt{</q>]]} & \text{if } \Gamma\ x\ =\ \texttt{page} \\ \mathcal{E}_{\mathsf{val}}[\![x\texttt{::}\omega\ v]\!]\ (\Gamma\ x) & \text{otherwise} \end{cases} \\
\mathcal{E}_{\mathsf{prop}}[\![x\texttt{::<q>}q\texttt{</q>}]\!]\ \Gamma &:= \begin{cases} \texttt{[[}\,x\texttt{::<q>}\ \mathcal{E}[\![q]\!]\ \Gamma\ \texttt{</q>]]} & \text{if } \Gamma\ x\ =\ \texttt{page} \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{E}_{\mathsf{prop}}[\![x\,.\,p\texttt{::}\phi]\!]\ \Gamma &:= \begin{cases} \texttt{[[}\,x\texttt{::<q>}\ \mathcal{E}_{\mathsf{prop}}[\![p\ \texttt{::}\ \phi]\!]\ \Gamma\ \texttt{</q>]]} & \text{if } \Gamma\ x\ =\ \texttt{page} \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{E}_{\mathsf{prop}}[\![x\texttt{::+}]\!]\ \Gamma &:= \texttt{[[}\,x\ \texttt{::+]]} \\
\mathcal{E}_{\mathsf{prop}}[\![x\texttt{::}\phi_1\ \texttt{||}\ \phi_2]\!]\ \Gamma &:= \mathcal{E}_{\mathsf{prop}}[\![x\ \texttt{::}\ \phi_1]\!]\ \Gamma\ \texttt{OR}\ \mathcal{E}_{\mathsf{prop}}[\![x\texttt{::}\ \phi_2]\!]\ \Gamma
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\texttt{[[Category:}c\texttt{]]}]\!]\ \Gamma &:= \mathcal{E}_{\mathsf{cat}}[\![c]\!] \\
\mathcal{E}[\![\texttt{[[}\iota\texttt{]]}]\!]\ \Gamma &:= \mathcal{E}_{\mathsf{ident}}[\![\iota]\!] \\
\mathcal{E}[\![\texttt{[[}p\texttt{::}\phi\texttt{]]}]\!]\ \Gamma &:= \mathcal{E}_{\mathsf{prop}}[\![p\texttt{::}\phi]\!]\ \Gamma \\
\mathcal{E}[\![q_1\ \texttt{OR}\ q_2]\!]\ \Gamma &:= \mathcal{E}[\![q_1]\!]\ \Gamma\ \texttt{OR}\ \mathcal{E}[\![q_2]\!]\ \Gamma \\
\mathcal{E}[\![q_1\ \texttt{AND}\ q_2]\!]\ \Gamma &:= \mathcal{E}[\![q_1]\!]\ \Gamma\ \texttt{AND}\ \mathcal{E}[\![q_2]\!]\ \Gamma
\end{aligned}
$$

We will take a deeper look at the elaboration of value selectors and property chains, as those are the most difficult.

If we look at how property chains are elaborated, we see that the first property in the chain is removed at each iteration, and the rest is recursively elaborated as a subquery. Since property chains are just syntactic sugar for a subquery (as shown in Section 2.4.4), the elaboration does not change the meaning of the query.

Example 5.3.3. Consider the following query:

```
[[Is located in.Has population::>1000000]]
```

Following our elaboration rules, we get:

```
[[Is located in::<q>[[Has population::>1000000]]</q>]]
```

Value selectors are elaborated differently based on the property's datatype. If the datatype is a page, we elaborate it to a subquery containing an identifier selector. We do this because identifier selectors and property selectors with pages are semantically equivalent, and so that our core language does not need a separate `page` datatype. It also serves as an example of how to elaborate other *complex* datatypes, such as records [2]. For the simple datatypes (`number`, `boolean` and `string`), we use $\mathcal{E}_{\mathsf{val}}$.

5.3.5 Type-soundness of elaboration

We have proven that the elaboration function is type-sound in Coq (Section 6.6.1). More specifically, we have proven that:

$$\forall\, q,\ \forall\, \Gamma,\ (\mathcal{E}[\![q]\!]\ \Gamma) \downarrow \rightarrow T\ (\mathcal{E}[\![q]\!]\ \Gamma)$$

Where T is a unary predicate that denotes whether a query is well-typed according to the typing rules given in Definition 4.4.3. In other words, we have proven that if the result of our elaboration is well-defined, then that resulting query is well-typed.

[2] https://www.semantic-mediawiki.org/wiki/Help:Type_Record

Chapter 6

Coq mechanization

6.1 Introduction

In this chapter, we will discuss how we mechanized our semantics in Coq and show how the most important definitions are encoded.

Everything from Chapter 4 and Chapter 5 has been mechanized in Coq. This includes the semantics of **Core Semantic MediaWiki** and the elaboration from **Surface Semantic MediaWiki** to **Core Semantic MediaWiki**. We have used this mechanization to prove that our elaboration is type-sound, and to prove a number of semantic equivalences of **Core Semantic MediaWiki** queries.

6.2 Coq-std++

We use the Coq-std++ (The Iris Development Team, 2023) library in our Coq mechanization, which provides a large number definitions and lemmas for well-known data structures. The most notable definitions that we use are:

- `gmap`, used to represent finite maps, and,
- `gset`, used to represent finite sets.

6.3 Values and identifiers

We encode names, identifiers and values (Definition 4.2.1) as shown in Listing 1 in a separate module named `Common`, since they are used in multiple places throughout the Coq development.

The definition `name` is simply an alias for a string. An identifier is encoded as a `Record` with two fields: ident_nspace containing the namespace of the identifier, and ident_art_name containing the article name of the identifier. Coq automatically introduces a function for each field of a record to retrieve that field. For example, if *i* is an identifier, we can write `ident_nspace` *i* to refer to the namespace of *i*. Values are encoded as an `Inductive`.

```
Module Common.
    Definition name := string.

    Record ident := Ident {
        ident_nspace : name;
        ident_art_name : name
    }.

    Inductive val :=
        | ValNat : nat → val
        | ValStr : string → val
        | ValBool : bool → val.
End Common.
```

Listing 1: The common definitions.

6.4 Database

We have mechanized the **Page** type (Definition 4.3.1) as a `Record` containing the fields cats and props as shown in Listing 2. The cats field is a finite set of names encoded as a `gset`, whereas the props field is a finite map going from names to sets of values, and is encoded as a `gmap`. The values in the database are encoded as a `sum` type.

The database function (Definition 4.3.2) is simply a map from `ident` to `page`.

```
Module Database.
    Definition val := sum Common.val Common.ident.

    Record page := Page {
        cats  : gset Common.name;
        props : gmap Common.name (gset val);
    }.

    Definition database := gmap Common.ident page.
End Database.
```

Listing 2: The definition of **Page** and **Database**.

6.5 Core Semantic MediaWiki

We use a number of Inductive definitions to represent the abstract syntax trees of **Core Semantic MediaWiki** (Definition 4.2.1) as shown in Listing 3.

The with keyword creates a mutually inductive definition. This is required for the mechanization of subqueries. We cannot use two separate Inductive definitions, because they would not be able to refer to each other.

The semantics (Definition 4.5.3) are implemented using a mutually recursive Fixpoint and are shown in Listing 3. A fixpoint is similar to a regular definition, but is able to call itself. To be able to formalize subqueries, the semantics need to be recursive.

The bool_decide function is a Coq-std++ definition that takes a (decidable) Prop and turns that into a boolean. This is required, because filter only works on decidable propositions, and Prop in general is not. It is not possible to change the return type of eval_selector to Prop and add it to the Decision typeclass, because of its mutual dependency on eval.

The semantics on paper are very close to the semantics in Coq. This makes it easy to verify that the Coq implementation indeed constitute the semantics given in this thesis on paper.

6.6 Surface Semantic MediaWiki

The abstract syntax of **Surface Semantic MediaWiki** is encoded in Coq using a number of Inductive definitions and is given in Listing 4.

The query and prop_sel definitions need to be mutually inductive, because of subqueries. The inductive definitions directly encode Definition 5.2.2.

6.6.1 Type-soundness of elaboration

We have used our Coq formalization to prove that our elaboration is type-sound (Section 5.3.5). That is, if the result of the elaboration is well-defined, then the result is well-typed. We have encoded this in Coq as follows:

```
Lemma elaborate_Some_typecheck
    (Γ : typing) (q : Surface.query) (q' : Core.query) :
    elaborate Γ q = Some q' → typecheck Γ q'.
```

Because the elaborate function operates on a *mutually* Inductive type (query), we have to use mutual induction for this proof. For induction on non-mutual inductive types, the induction principle that Coq automatically generates is often sufficient. However, for mutually inductive types, this is not the case. Coq only automatically generates non-mutual induction principles. This means that the inductive proof will get stuck on the `PropSel` case, since we know nothing about the `prop_sel` parameter. To solve this, we use the `Scheme` command to generate a mutual induction principle manually, and use that to prove `elaborate_Some_typecheck`.

```
Module Core.
    Inductive comp := CompEq | CompNeq | CompGt | CompLt.
    Inductive query :=
        | Selector : selector → query
        | Disjunction : query → query → query
        | Conjunction : query → query → query
    with selector :=
        | Category : Common.name → selector
        | Namespace : Common.name → selector
        | Existence : Common.name → selector
        | Subquery : Common.name → query → selector
        | Article : comp → Common.name → selector
        | Value : comp → Common.name → Common.val → selector.

    Fixpoint eval (q : Core.query) (d : Database.database) :
        gset Common.ident
    := match q with
        | Core.Selector s         =>
          dom (filter (λ i_di, eval_selector s i_di.1 i_di.2 d) d)
        | Core.Disjunction q1 q2 => eval q1 d ∪ eval q2 d
        | Core.Conjunction q1 q2 => eval q1 d ∩ eval q2 d
        end
    with eval_selector
        (s : Core.selector)
        (i : Common.ident)
        (di : Database.page)
        (d : Database.database)
    : bool :=
        match s with
        | Core.Category x   => bool_decide (x ∈ Database.cats di)
        | Core.Namespace x  => bool_decide
            (x = Common.ident_nspace i)
        | Core.Existence x  => bool_decide
            (set_inhabited $ Database.props di !!! x)
        | Core.Subquery x q => bool_decide
            (set_inhabited
                (Database.props di !!! x ∩ set_map inr (eval q d)))
        | Core.Article ω x  => bool_decide
            (eval_art_name_comp ω (Common.ident_art_name i) x)
        | Core.Value ω x v' => bool_decide
            (set_Exists
                (λ v, eval_val_comp ω v v')
                (Database.props di !!! x))
        end.
End Core.
```

Listing 3: The abstract syntax and the semantic function of **Core Semantic MediaWiki**.

```
Module Surface.
    Inductive comp :=
        | CompEq | CompNeq
        | CompGt | CompLt
        | CompGte | CompLte.
    Inductive ref :=
        | Ident : Common.ident → ref
        | ArtName : Common.name → ref.
    Inductive prop :=
        | SingletonProperty : Common.name → prop
        | PropertyChain : Common.name → prop → prop.
    Inductive cat_sel :=
        | Category : Common.name → cat_sel
        | CatSelDisj : cat_sel → cat_sel → cat_sel.
    Inductive ident_sel :=
        | Namespace : Common.name → ident_sel
        | IdentComp : comp → ref → ident_sel
        | IdentSelDisj : ident_sel → ident_sel → ident_sel.
    Inductive query :=
        | CatSel : cat_sel → query
        | IdentSel : ident_sel → query
        | PropSel : prop → prop_sel → query
        | Disjunction : query → query → query
        | Conjunction : query → query → query
    with prop_sel :=
        | Wildcard : prop_sel
        | Subquery : query → prop_sel
        | Value : comp → string → prop_sel
        | PropSelDisj : prop_sel → prop_sel → prop_sel.
End Surface.
```

Listing 4: The abstract syntax of **Surface Semantic MediaWiki**.

Chapter 7

Related Work

In this chapter, we discuss related work. Since not much academic work has been done formalizing Semantic MediaWiki in particular, this section will also look into formalizations of other (query) languages, such as SQL.

7.1 SQL formalizations

Guagliardo and Libkin, 2017 give the formal semantics of a basic fragment of SQL. In particular, they include the handling of NULL values in their fragment, which is non-trivial. Using their semantics, they show that the formalized fragment, called *basic SQL*, has the same expressive power as relational algebra extended with bag semantics and NULLs. Furthermore, they demonstrate that the common believe that SQL queries require three-valued logic (`true`, `false` and `unknown`) is incorrect. They do this by providing a translation from two-values semantics, where **true** and **unknown** are conflated, to three-valued semantics. We do not consider the handling of NULL values in our semantics, because they do not exist in Semantic MediaWiki.

Chu et al., 2017 give denotational semantics for a large fragment of SQL, that includes bags, correlated subqueries, aggregations and indexes. They also implement their semantics in Coq as part of COSETTE. COSETTE is a tool that can be used to prove the equivalence of SQL rewrite rules. The power of COSETTE is demonstrated through 23 equivalence proofs.

Benzaken and Contejean, 2019 give executable semantics of a realistic fragment of SQL, called SQL_{Coq}, and also mechanize these semantics it in Coq. Similar to our semantics, they started with set-theoretic semantics. Only later in the development were bag semantics added. According to the authors, the real challenge in defining semantics for SQL are correlated queries, and especially SQL's management of expressions and environments. Using their semantics, they were able to prove that SQL_{Coq} is equivalent to relational algebra.

7.2 Verifications of query compilers

*Q*cert* (Auerbach et al., 2017) is a platform for the development and verification of query compilers. It allows you to describe and reason about a compiler in Coq. The compiler is then extracted from Coq using Coq's extraction mechanism. Q*Cert supports the implementation of compilers with a number of different front-ends, such as SQL, OQL or JRule. It has the ability to compile these different front-ends, to a number of back-ends, such as Java, JavaScript or IBM's Cloudant.

DBCert (Benzaken et al., 2022) is a mechanically verified optimizing compiler implemented in the Coq proof assistant that goes from standard SQL queries to imperative JavaScript. This JavaScript can then be used to query an in-memory database through Node.js. DBCert works by first parsing standard SQL into a canonical form of SQL called SQL_{Coq}, which in turn is compiled to an extended version of relational algebra called SQL_{Alg}. SQL_{Alg} is translated to NRA^{e}, which is where query optimizations can be performed. Finally, NRA^{e} is translated using different intermediate languages, to the imperative language Imp, which is used for generating the JavaScript code. Every step of the compilation from SQL_{Coq} to Imp is formally verified using Coq. DBCert also handles nested queries (subqueries), which is challenging because of correlation between the queries, where a nested query uses a variable from the outer query. This is not an issue for Semantic MediaWiki, because it does not have the notion of variables.

7.3 Formalizations of other query languages

While much academic work focuses on formalizing SQL, there are also formalizations of other query languages.

Pérez et al., 2009 provide compositional semantics for a fragment of SPARQL, and use it to provide several rewrite rules that may be used for optimizing queries.

Díaz et al., 2020 give the *first* mechanized formalization of GraphQL, called GraphCoQL. As the name suggests, the formalization is implemented in the Coq proof assistant. They demonstrate GraphCoQL by studying the normalization process of GraphQL queries proposed by Hartig and Pérez, 2018. More specifically, they define a property in GraphCoQL about whether a query is in *normal form* and define and verify a normalization procedure using their formalization.

7.4 Semantic MediaWiki

Papers talking about formalizing Semantic MediaWiki are unfortunately few and far between. As far as we know, there is only one other paper that worked on formalizing Semantic MediaWiki.

Bao et al., 2008 give semantics for both the modelling language (the language that is used to insert new data into a wiki) and the query language of Semantic MediaWiki.

They formalize the query language of Semantic MediaWiki by translating queries into logic programs. This translation can be used to transform any Semantic MediaWiki query q to a logic program, which, when executed, will respond with the pages that match q. They also provide a translation from the data model to logic programs. Using their semantics, Bao et al., 2008 show that query answering in Semantic MediaWiki is P-complete for the number of annotations in the database, meaning that it can be solved in polynomial time, or logarithmic if subqueries are not considered. It should be noted that the formalization does not consider data types appropriately, as they only consider the data type page. However, they argue that the semantics can easily be extended incorporate data types.

Furthermore, Bao et al., 2008 present a *model-theoretical* semantics for the modelling language used in Semantic MediaWiki, used to insert new data into the database. Model-theoretical semantics is an approach that is used in formal logic to provide a precise interpretation of a language. They define an *interpretation* of a Semantic MediaWiki wiki as a tuple, W, that contains the semantic data stored in the wiki. An interpretation is a *model* of W if it satisfies a number of conditions that reflect the behaviour of Semantic MediaWiki's modelling language. For example, if a page contains the annotation `[[Category:City]]`, an interpretation must establish that the page is then part of the `City` category. They use their semantics to reason about the inference capabilities that Semantic MediaWiki offers through a number of entailment rules. Furthermore, they show that these entailment rules are sound and complete, and they show that inference in Semantic MediaWiki is tractable.

Finally, Bao et al., 2008 provide a number of improvements that can be made to the modelling and query language of Semantic MediaWiki. Some of these improvements have already been implemented in later versions, such as inverse properties (Semantic MediaWiki contributors, 2018b).

Chapter 8

Conclusions and future work

In this thesis, we have formalized a fragment of the query language of Semantic MediaWiki and have provided a mechanized semantics in Coq. We did this by defining the semantics for a minimal language that captures all the features of our fragment of Semantic MediaWiki, called Core Semantic MediaWiki, and by providing a translation from regular Semantic MediaWiki queries to Core Semantic MediaWiki. Furthermore, we used our semantics to prove a number of semantic equivalences.

Future work includes defining the semantics for more of the features of Semantic MediaWiki, such as:

- Inverse properties (Semantic MediaWiki contributors, 2018b)
- Like and not-like comparators (Semantic MediaWiki contributors, 2023)
- Full-text search (Semantic MediaWiki contributors, 2018a)
- Inferencing (Semantic MediaWiki contributors, 2020a)
- Additional datatypes (Semantic MediaWiki contributors, 2020b)

Other future work could be the verification of the query compiler for Semantic MediaWiki. For example, by using the semantics of SQL_{Coq} (Benzaken & Contejean, 2019) and our semantics of Semantic MediaWiki to prove that all compiled queries are equivalent to the respective source query. Finally, one could do the same for other supported back-ends (SPARQL and Elasticsearch) and in the process show that the different back-ends give equivalent results.

Bibliography

Appel, A. W., & Blazy, S. (2007). Separation logic for small-step cminor. In K. Schneider & J. Brandt (Eds.), *Theorem proving in higher order logics, 20th international conference, tphols 2007, kaiserslautern, germany, september 10-13, 2007, proceedings* (pp. 5–21, Vol. 4732). Springer. https://doi.org/10.1007/978-3-540-74591-4_3

Auerbach, J. S., Hirzel, M., Mandel, L., Shinnar, A., & Siméon, J. (2017). Q*cert: A platform for implementing and verifying query compilers. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, & D. Suciu (Eds.), *Proceedings of the 2017 ACM international conference on management of data, SIGMOD conference 2017, chicago, il, usa, may 14-19, 2017* (pp. 1703–1706). ACM. https://doi.org/10.1145/3035918.3056447

Bao, J., Ding, L., & Hendler, J. (2008). Knowledge representation and query in Semantic MediaWiki: A formal study.

Benzaken, V., & Contejean, E. (2019). A Coq mechanised formal semantics for realistic SQL queries: Formally reconciling SQL and bag relational algebra. In A. Mahboubi & M. O. Myreen (Eds.), *Proceedings of the 8th ACM SIGPLAN international conference on certified programs and proofs, CPP 2019, cascais, portugal, january 14-15, 2019* (pp. 249–261). ACM. https://doi.org/10.1145/3293880.3294107

Benzaken, V., Contejean, E., Hachmaoui, M. H., Keller, C., Mandel, L., Shinnar, A., & Siméon, J. (2022). Translating canonical SQL to imperative code in Coq. *Proc. ACM Program. Lang.*, *6*(OOPSLA1), 1–27. https://doi.org/10.1145/3527327

Chu, S., Weitz, K., Cheung, A., & Suciu, D. (2017). HoTTSQL: Proving query rewrites with univalent SQL semantics. In A. Cohen & M. T. Vechev (Eds.), *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation, PLDI 2017, barcelona, spain, june 18-23, 2017* (pp. 510–524). ACM. https://doi.org/10.1145/3062341.3062348

Coq Development Team. (2021). The Coq proof assistant reference manual. https://coq.inria.fr/doc/

Díaz, T., Olmedo, F., & Tanter, É. (2020). A mechanized formalization of GraphQL. In J. Blanchette & C. Hritcu (Eds.), *Proceedings of the 9th ACM SIGPLAN* *international conference on certified programs and proofs, CPP 2020, new orleans, la, usa, january 20-21, 2020* (pp. 201–214). ACM. https://doi.org/10.1145/3372885.3373822

Floyd, R. W. (1967). Assigning meanings to programs. *Mathematical Aspects of Computer Science*, *19*, 19–32. https://doi.org/10.1090/psapm/019/0235771

Guagliardo, P., & Libkin, L. (2017). A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, *11*(1), 27–39. https://doi.org/10.14778/3151113.3151116

Hartig, O., & Pérez, J. (2018). Semantics and complexity of GraphQL. In P. Champin, F. Gandon, M. Lalmas, & P. G. Ipeirotis (Eds.), *Proceedings of the 2018 world wide web conference on world wide web, WWW 2018, lyon, france, april 23-27, 2018* (pp. 1155–1164). ACM. https://doi.org/10.1145/3178876.3186014

Jung, R., Jourdan, J., Krebbers, R., & Dreyer, D. (2018). RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, *2*(POPL), 66:1–66:34. https://doi.org/10.1145/3158154

Krebbers, R. J. (2015). *The C standard formalized in Coq* [Doctoral dissertation].

Lassila, O., & Swick, R. R. (1999). Resource description framework (RDF) model and syntax specification. http://www.w3.org/TR/1999/PR-rdf-syntax-19990105

Leroy, X. (2009). A formally verified compiler back-end. *J. Autom. Reason.*, *43*(4), 363–446. https://doi.org/10.1007/s10817-009-9155-4

MediaWiki contributors. (2022). Manual:Page title [Online; accessed 18-April-2023]. https://www.mediawiki.org/w/index.php?title=Manual:Page_title&oldid=5609477

Nielson, H. R., & Nielson, F. (1992). *Semantics with applications - a formal introduction*. Wiley.

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, *34*(3), 16:1–16:45. https://doi.org/10.1145/1567274.1567278

Semantic MediaWiki contributors. (2018a). Help:full-text search [Online; accessed 31-May-2023]. https://www.semantic-mediawiki.org/w/index.php?title=Help:Full-text_search&oldid=64477

Semantic MediaWiki contributors. (2018b). Help:inverse properties [Online; accessed 31-May-2023]. https://www.semantic-mediawiki.org/w/index.php?title=Help:Inverse_properties&oldid=62695

Semantic MediaWiki contributors. (2019). Help:user manual [Online; accessed 31-May-2023]. https://www.semantic-mediawiki.org/w/index.php?title=Help:User_manual&oldid=68119

Semantic MediaWiki contributors. (2020a). Help:inferencing [Online; accessed 27-May-2023]. https://www.semantic-mediawiki.org/w/index.php?title=Help:Inferencing&oldid=73704

Semantic MediaWiki contributors. (2020b). Help:list of datatypes [Online; accessed 31-May-2023]. https://www.semantic-mediawiki.org/w/index.php?title=Help:List_of_datatypes&oldid=74038

Semantic MediaWiki contributors. (2020c). Semantic MediaWiki 1.0.0 [Online; accessed 24-May-2023]. https://www.semantic-mediawiki.org/w/index.php?title=Semantic_MediaWiki_1.0.0&oldid=74787

Semantic MediaWiki contributors. (2023). Help:search operators [Online; accessed 31-May-2023]. https://www.semantic-mediawiki.org/w/index.php?title=Help:Search_operators&oldid=84481

Slind, K., & Norrish, M. (2008). A brief overview of HOL4. In O. A. Mohamed, C. Muñoz, & S. Tahar (Eds.), *Theorem proving in higher order logics* (pp. 28–32). Springer Berlin Heidelberg.

The Iris Development Team. (2023). Coq-std++. https://gitlab.mpi-sws.org/iris/stdpp

van Wezel, M. (2023). *Formalizing the query language of Semantic MediaWiki* (Version 1.0.0). Zenodo. https://doi.org/10.5281/zenodo.8086728

Wenzel, M., Paulson, L. C., & Nipkow, T. (2008). The Isabelle framework. In O. A. Mohamed, C. Muñoz, & S. Tahar (Eds.), *Theorem proving in higher order logics* (pp. 33–38). Springer Berlin Heidelberg.