Frog: Functions for ontologies

An extension for the OTTR-framework

Marlen Jarholt

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

Frog: Functions for ontologies

An extension for the OTTR-framework

Marlen Jarholt

Frog: Functions for ontologies

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Reasonable Ontology Templates (OTTR) is a language making it possible to compose parameterised modelling patterns, known as templates, that we can instantiate to produce an RDF graph and OWL ontologies. Hence, removing the repetitive and time-consuming processes of producing RDF graphs over a domain. Using OTTR offers several benefits compared to pure RDF: the Don't repeat yourself (DRY) principle, better abstraction, uniform modelling, and separation of design and content. To instantiate templates, we often use bOTTR and tabOTTR, which extract data from a tabular file or data sources. OTTR templates can not perform calculations on argument terms. Consequently, we need to perform calculations over terms before instantiating them. Therefore, the sources that bOTTR and tabOTTR extract data from need to perform necessary calculations. Each source type has different means to create calculations, thus, resulting in several ways of manipulations for the same calculation. We believe that making it possible to have one uniform means of performing calculations in templates will strengthen OTTR's aforementioned benefits. Therefore, in this thesis, we design and implement a programing language, Frog, that can perform manipulations inside templates, which aims to integrate with OTTR seamlessly. Moreover, we evaluate if including a language like Frog into OTTR enhances the acclaimed benefits.

Acknowledgements

I want to thank my supervisor, Leif Harald Karlsen, for his excellent advice and guidance throughout my thesis. I would like to thank my friends and co-students for helpful discussions and for letting me ramble about Frog. Especially, I would like to thank Simen Fonnes for providing me with feedback and proofreading my thesis. Finally, I want to thank my parents, Trygve and Irene Jarholt, and my brother, André Jarholt, for their continuous support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

In 2001 Tim Bernes-Lee published his vision of the semantic web, an extension of the World Wide Web [4]. The motivation behind the semantic web was to give data on the web a formal description and a well-defined meaning. Thus, the data in the semantic web is easily machine-readable, enabling better cooperation between computers and humans [4]. To create these formal descriptions and well-defined meanings, the World Wide Web Consortium (W3C) has created a set of standards, which we refer to as *semantic technologies*. These technologies provide a formal description of terms, relationships and concepts within knowledge domains. Moreover, semantic technologies creates or works on data structured as linked data. Mauro and Tiziana summarised link data as a means to connect, expose and share web data utilising identifiers [12].

In the case of the semantic web, a connection is an identifier connecting one resource to another resource, describing the relationship the first identifier has to the second. Therefore, a connection consists of three elements: *subject*, *predicate*, and *object*. Where the predicate is a connection from the subject to the object, describing the subject's relationship to the object. A subject that connects to an object through a predicate is called a *triple*. The technology used to describe and structure these triples are the Resource Description Framework (RDF) [9]. We often refer to a set of triples in an RDF document as an *RDF graph*.

Building an RDF graph often involves creating a large number of triples. Regularly an RDF graph consists of triples that have the same structure. For instance, an RDF graph with data about persons would possibly contain triples stating a person's age, name, social security number, parents, family members, and primary residence. Writing out these triples for one person does not take a significant amount of time. However, creating a graph that contains 100 persons, where every person has at least one parent and one family member, would at least require 600 triples. Writing these triples manually in RDF would be a tedious job and containing a large amount of repetition.

Reasonable Ontology Templates (OTTR) [37] addresses these issues by making it possible to compose templates which contain parametrised modelling patterns. These templates can therefore encapsulate domains' structures. OTTR uses templates to create

RDF graphs; hence we can consider OTTR a macro language for RDF. To create the RDF graph, OTTR expands instances. An instance refers to a template and has arguments; when expanded, OTTR replaces the parameter variables in the affiliated template's parametrised modelling pattern with these arguments. Thus, we can construct a template for a domain and create an RDF graph by instantiating instances that OTTR expands. Additionally, OTTR provides two means for instantiating instances from tabular files and different data sources: tabOTTR and bOTTR. tabOTTR [23] transforms tabular files into instances, while bOTTR [22] allows us to create mappings that extracts data from numerous types of sources, for instance, through SQL and SPARQL[1] queries.

Constructing the aforementioned RDF graph containing data about 100 persons would now only require us to create one template that encapsulates our model of a person and one instance per person. It can still be time-consuming to produce these 100 instances. However, if the data already exists in a tabular file or data source, e.g. in a CSV file, building an RDF graph would now only require one template and one bOTTR mapping creating instances from the CSV file. As a result, utilising OTTR, compared to writing triples manually, offers several benefits, such as the *Don't repeat yourself (DRY)* principle, *better abstraction*, *uniform modelling* and *separation of design and content* [38].

Nevertheless, the previously mentioned template used to create an RDF graph of persons from the CSV file may require additional information that the file naturally would not contain, such as the URI, to identify each person uniquely. The social security number could, for instance, be a natural part of a URI to identify a person uniquely. However, we can not create URIs inside the templates since OTTR templates only encapsulate patterns and not perform calculations. Consequently, the creator of the CSV mapping needs to produce the URIs in the mapping. We argue that being able to create URIs inside templates improves abstraction; since templates also can abstract the logic over their values. The templates producing URIs would be especially beneficial if we use multiple files to create the instances. If we have 50 CSV files, we must create 50 bOTTR mappings containing URI calculations, which is repetitious. However, if we could create the calculation in the person template, we only need one calculation, thus, enforcing the DRY principle.

Moreover, creating an RDF graph with OTTR may require extracting data from different sources and tabular files. OTTR allows users to extract data from tabular formats with tabOTTR and by querying over sources by creating JDBC[2] or SPARQL queries. Each of these formats has different approaches for manipulating data. Consequently, we need to create one type of calculation for each source type. Placing the calculation in the templates will only require one uniform means of calculation. Thus, strengthening OTTR's benefit of uniform modelling.

Additionally, by making it possible to manipulate terms in the templates, we can move some RDF specific tasks, such as creating URIs in tabular files or mappings, into tem-

[1]SPARQL is a query language over RDF graphs.

[2]Java Database Connectivity (JDBC) is an API data used to query over sources through different query languages, such as SQL and PostgreSQL. `https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/`

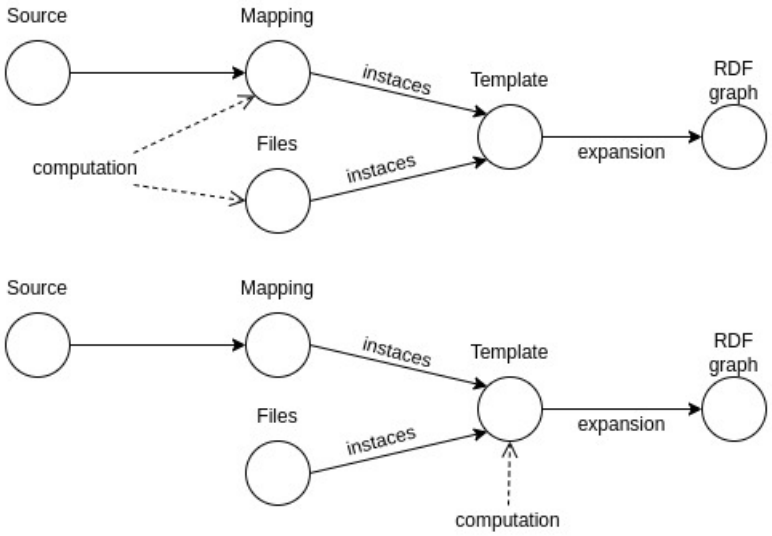

Figure 1.1: The first graph shows a natural workflow utilises OTTR, where the computations are preformed in tabular files or by mappings. The second graph shows a natural workflow using OTTR with Frog, where the computations are preformed in templates.

plates. Hence, templates can become an abstraction for RDF specific elements, which we argue imporves the better abstraction benefit.

There are several other use-cases where manipulating terms inside templates will be beneficial, such as storing information about one domain that may have numerous representations for the same thing. An example of such a domain weather and temperature data. Imagine that we want to store information about weather temperatures. In this case, we may want to use several sources of data, both American and European. As a result, some sources would contain degrees in Celcius and others in Fahrenheit. We, however, want our RDF graph to only contain the degrees in Celcius. Now, without a means of performing calculations in the templates, we must perform the calculations in every tabular file and mapping that uses the Fahrenheit scale. Having the opportunity to perform this calculation in the templates will be beneficial as we only need to do it once instead of in all the tabular files and mappings with Farhenheit degrees. Additionally, this template encapsulates our model, which includes that the degrees should be in Celcius. Without a means of performing calculations in the templates, OTTR must assume that the degrees are on the correct scale.

1.2 Problem statment and scope

From the discussions above, we argue that the possibility to modify and perform calculations on values in templates, also called terms, will strengthen OTTR's benefits. Therefore, in this thesis, we present and create a way to modify terms within templates. Thus, making it possible to move calculations from tabular files and mappings into the templates themselves, as Figure 1.1 depicts. Our proposed solution is to make a small programming language, which seamlessly integrates with OTTR, named *Functions for ontologies* abbreviated to *Frog*.

In this thesis, we investigate how we can design Frog to be a functional programming language that seamlessly integrates with OTTR. For Frog to seamlessly integrate with OTTR, Frog must not conflict with OTTR's semantics while being easy and natural to place inside a template's parametrised modelling pattern. Moreover, we examine and discuss whether our claims that the inclusion of Frog into OTTR enhances OTTR's following benefits:

- Don't repeat yourself (DRY) principle,
- better abstraction,
- uniform modelling, and
- separation of design and content.

In this thesis, we alos create an implementation of Frog into the existing reference implementation of OTTR, *Lutra*[3]. We create this implementation to be able to evaluate if Frog improves the acclaimed benefits and to test that Frog's design performs sufficiently in practice. Hence, the aim ofthe implementation provided in this thesis is to be a proof of concept and not an efficient implementation.

1.3 Outline

This thesis is structured with the following chapters:

- **Chapter 2: Functional Programming**
  Frog will be a functional programming language. Therefore, Chapter 2 introduces functional programming and some of its principles. We introduce lambda calculus which is considered the first functional programming language, and a typed version of lambda calculus: simply typed lambda calculus. Additionally, we discuss different evaluation strategies for functional programs.
- **Chapter 3: Semantic Web & OTTR**
  In this chapter, we introduce OTTR's concepts and semantics to be able to determine how we should design Frog. Additionally, we introduce RDF, the semantic technology OTTR is a macro language for, and SHACL, a semantic technology OTTR applies. This section also introduces the semantic technology SPARQL, a query language over RDF grafs that we will use in our implementation Frog.
- **Chapter 4: Design**
  This chapter discusses the design of Frog and investigates how we must design Frog to integrate seamlessly with OTTR. This chapter contains the formal description of Frog.
- **Chapter 5: Implemntation**
  In this chapter, we discuss our Frog implementation into OTTR's reference implementation, Lutra.

[3] `https://gitlab.com/ottr/lutra/lutra`

- **Chapter 6: Discussion**
  This chapter discusses whether Frog has improved our claimed benefits in OTTR: the DRY principle, better abstraction, uniform modelling, and separation of design and content. In other words, this chapter discusses whether Frog improves the OTTR framework by making it possible to manipulate terms inside the templates. Additionally, we discuss and argue for some of the choices in the implementation and design of Frog.

- **Chapter 7: Related Work**
  In this chapter, we discuss related work to Frog. We introduce several technologies that we can utilise to manipulate RDF terms. Moreover, we discuss why these technologies do not fulfil our criteria for a term manipulation that seamlessly integrates with the OTTR framework.

- **Chapter 8: Conclusion**
  In the final chapter, we summarise our findings and provide possible future work regarding OTTR and Frog.

Chapter 2

Functional Programming

Frog will be a functional programming language. Hence, we shortly introduce functional programming in this chapter. We start, in Section 2.1, by introducing Lambda calculus, a formal system with the aim of encapturing intuitions on functions and is considered the first functional programming language [16]. Moreover, in Section 2.1.4, we introduce combinatory logic. In Section 2.2 we introduce simply typed lambda calculus, which is a typed version of lambda calculus. Lastly, Section 2.3 discusses evaluation strategies, and Section 2.4 introduces functional programming in the more general case and its principles.

2.1 Lambda Calculus

Lambda calculus is a collection of formal systems for expressing computation and is the foundation for functional programming. Alonzo Church introduced lambda calculus in the 1930s and published his first book on this topic in 1941 [7]. Lambda calculus is a universal model of computation that can express any computable function, thus, being Turing complete. We base the following section on Rojas's paper on lambda calculus [40].

2.1.1 Definitions and notations

Lambda calculus consists of three primitives symbols: λ, (, and), in addition to a set of variables. The fundamental concept in lambda calculus is a lambda term, also called an expression. A lambda term is either a:

- *Variable*: A name which is a placeholder for a parameter. A variable x is in itself a lambda term.
- *Abstraction*: If we have a term M and an variable x, then we also have the term $\lambda x.M$ where we say that x is bound in M. The abstraction $\lambda xy.M$ is an abbreviation of $\lambda x.\lambda y.M$.
- *Application*: If E_1 and E_2 are lambda terms, then (E_1E_2) is also a lambda term, where we say that E_2 is applied to E_1. λ-calculus' application terms are left-associative, which means that $E_1E_2E_3 \equiv (E_1E_2)E_3$. We omit parenthesis when the meaning is clear.

Lambda calculus builds up its λ-terms using these three expressions, which Figure 2.1 defines in BNF-syntax.

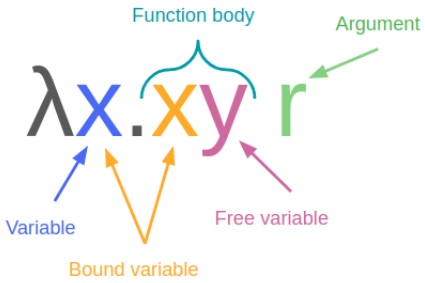

Figure 2.2: Visualization of the different concepts in a λ-function.

$$\begin{array}{lcll} \lambda\text{-term} & := & \text{x} & \text{(variable)} \\ & | & \lambda x.M & \text{(abstraction)} \\ & | & E_1 E_2 & \text{(application)} \end{array}$$

Figure 2.1: λ-calculus' syntax in BNF.

All functions in λ-calculus are first-class values, which means that a lambda function can take a λ-function as an argumnent and return λ-functions. Figure 2.2 illustrates simple visualization of the different concepts in a λ-function.

2.1.2 Bound and Free variables

As previously mentioned, the abstraction contains *bound variables*. The λ binds the variables stated before the function body to the variables with the same symbol in the function body. We consider the variables that both occur in the function body and the λ section before the function body as the bound variables. An example of a bound variable is the variable x, in $\lambda x.xy$ since x occurs both in the λ section of the function and the function body.

Free variables are the the variables in a term that is not bound by the abstraction. Formally, if a term E_1 containing a set of variables, V_1, and a set of bound variables, B_1, then the free variables in E_1 are $F_1 \equiv V_1 \setminus B_1$. We can extend this general rule further, by adding another term E_2 containing a set of variables, V_2, and a set of bound variables, B_2; then the free variables in the term E_1E_2, $F_1F_2 \equiv (V_1 \setminus B_1) \cup (V_2 \setminus B_2)$. In other words, the set of free variables in the term E_1E_2 is the union of the free variables in E_1 and E_2. In our previous λ function, $\lambda x.xy$, we consider the variable y as a free variable as it occurs in the function body but is not present in the λ binding.

2.1.3 Conversion and Reduction

We introduce two forms for conversion and reduction in λ-calculus, namely α-conversion and β-reduction. Firstly, *α-conversion* ($\rightarrow_\alpha$) is the procedure of transforming a λ-term into another term that is α-equivalent and that yields the same result for a set of arguments. Two λ-terms are *α-equivalent* iff the only dissimilarity in the two terms are the variables. For instance, $\lambda ab.ab$ is α-equivalent to the term $\lambda xy.xy$. α-conversion

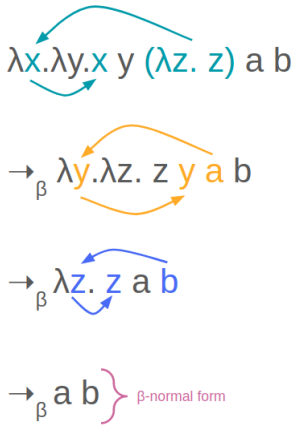

Figure 2.3: An example of β-reduction.

changes a variable in the λ binding and the function body with another variable. Importantly, we can not change a variable into variable that allready exists in the term, as this may yield different results for a set of arguments. Therefore, $\lambda x.xy \rightarrow_\alpha \lambda z.zy$ is a valid α-conversion, while $\lambda x.xy \rightarrow_\alpha \lambda y.yy$ is not. Notably, we can only change the variables in the same abstraction. Hence, $\lambda x.\lambda x.x \rightarrow_\alpha \lambda x.\lambda y.y$ is a valid α-conversion while $\lambda x.\lambda x.x \rightarrow_\alpha \lambda y.\lambda x.y$ is not.

Moreover, *β-reduction* ($\rightarrow_\beta$) is a procedure to reduce terms by substituting its arguments into the function body. In other words, removing the second term, E_2 from the definition of application, by substituting it into the first term's body, E_1. For instance, if E_2 is a and E_1 is $\lambda x.x$ then $E_1E_2 \equiv (\lambda x.x)a$ which we can performe a β-reduction on $(\lambda x.x)a \rightarrow_\beta a$. Figure 2.3 illustrates a set of β-reductions on the term $\lambda x.\lambda y.xy(\lambda z.z)\ a\ b$. When there are no more possible β-reductions to perform on a term, we say that the term is on *β-normal form*. A term on β-normal form is either a set of variables or an abstraction. However, a term may never reach β-normal form, meaning that attempting to perform β-reductions on a term until it reaches β-normal form may not terminate. An example of a term that does not terminate when performing β-reductions is the self-applicate function applied on itself, $(\lambda f.ff)\lambda f.ff$; since one β-reductions this term results in the same term. Futhermore, we say that a term, M, is *β-equal* to another term, N, if finite β-reductions on the term M results in the term N. This is often denoted with $M =_\beta N$. In Figure 2.3 the term $\lambda x.\lambda y.xy\ (\lambda z.z)\ a\ b$ is β-equal to all the terms beneath.

2.1.4 Combinatory logic

Combinatory logic in computer science is a theoretical theory of computation introduced by Schönfinkel [41] and Curry [8]. The main idea of combinatory logic is to introduce some base *combinators* or constants that we combine to create expressions. Examples of such combinators in combinatory logic are **S**, **K**, **I**, **B**, and **C**; we can express the combinators **I**, **B**, and **C** from **S** and **K**. Combining combinators with an infinite set of variables can express any other high-order functions. Combinatory logic is closely associated with λ-calculus, and we can create translators from a λ-terms to a set of combinators, often named transformation, bracket-abstraction algorithm or abstract elimination. Schönfinkel, for instance, introduced a bracket-abstraction algorithm on the **S**, **K**, **I**, **B**, and **C**

$E :=$	x	(variable)
$\mid$	P	(a combinator)
$\mid$	E_1E_1	(application, where E_1 and E_2 are combinatoric terms)

Figure 2.4: How to build up combinatoric terms in BNF-syntax.

combinators [5]. In this section, we present the **SK** combinators and Lachowski bracket abstraction [28] for the **SK** combinators.

The **K** combinator that takes in two arguments and always returns the first argument, defined by:

$$((Kx)y) = x \text{ or } (Kxy) = x$$

Written in λ-calculus, the **K** function would look like this:

$$\lambda xy.x$$

Furthermore, we have the combinator **S**, which takes in three arguments and applies the last argument on the two first arguments:

$$(Sxyz) = (xz(yz))$$

Written in λ-calculus, the **S** function would look like this:

$$\lambda xyz.xz(yz)$$

From these combinators and a variables, we can build up combinatoric terms as shown in Figure 2.4.

Lachowski bracket-abstraction algorithm [28]:

1. $[x] = x$, for every variable x
2. $[MN] = [M][N]$
3. $[\lambda x.M] = \lambda^* x.M$
4. $\lambda^* x.x = \mathbf{SKK}$
5. $\lambda^* P = \mathbf{K}P$ if x is a free variable in P.
6. $\lambda^* MN = \mathbf{S}(\lambda^* M)(\lambda^* N)$ if x is a bound variable in MN.

The following example shows how to transform the expression $\lambda x.\lambda y.yyx$ into **SK** combinatory logic:

$[\lambda.xy.yyx] =_3 \lambda^*x.[\lambda y.yyx]$
$=_3 \lambda^*x.\lambda^*y.yyx$
$=_6 \lambda^*x.\mathbf{S}(\lambda^*y.yy)(\lambda^*y.x)$
$=_5 \lambda^*x.\mathbf{S}(\lambda^*y.yy)(\mathbf{K}x)$
$=_6 \lambda^*x.\mathbf{S}(\mathbf{S}(\lambda^*y.y)(\lambda^*y.y))(\mathbf{K}x)$
$=_4 \lambda^*x.\mathbf{S}(\mathbf{S}(\mathbf{SKK})(\lambda^*y.y))(\mathbf{K}x)$
$=_4 \lambda^*x.\mathbf{S}(\mathbf{S}(\mathbf{SKK})(\mathbf{SKK})(\mathbf{K}x)$
$=_6 \mathbf{S}(\lambda^*x.\mathbf{S}(\mathbf{S}(\mathbf{SKK})(\mathbf{SKK}))(\lambda^*x.(\mathbf{K}x))$
$=_5 \mathbf{S}(\mathbf{KS}(\mathbf{S}(\mathbf{SKK})(\mathbf{SKK}))(\lambda^*x.(\mathbf{K}x))$
$=_6 \mathbf{S}(\mathbf{KS}(\mathbf{S}(\mathbf{SKK})(\mathbf{SKK}))(\mathbf{S}(\lambda^*x.\mathbf{K})(\lambda^*x.x)))$
$=_5 \mathbf{S}(\mathbf{KS}(\mathbf{S}(\mathbf{SKK})(\mathbf{SKK}))(\mathbf{S}(\mathbf{KK})(\lambda^*x.x)))$
$=_4 \mathbf{S}(\mathbf{KS}(\mathbf{S}(\mathbf{SKK})(\mathbf{SKK}))(\mathbf{S}(\mathbf{KK})(\mathbf{SKK})))$

2.2 Simply Typed Lambda Calculus

Simply Typed Lambda Calculus ($\lambda^{\rightarrow}$), made by Alonzo Church in 1940 [7], is a type theory for lambda-calculus. In short, type theory is the academic study of type systems, which gives every term a type. Jackobs [19] states that the use of types is to classify values.

In $\lambda^{\rightarrow}$-calculus, we provide the λ-terms with types. There are two different approaches for providing variables with a type. Firstly, *typing à la Church* also named *explicitly typing*, where we state the type of the variable when introducing it. Secondly, *typing à la Curry*, which does not give types to variables when introduced but leaves them open. Often the typing in typing à la Curry is found through a search process, including some qualified guessing [31]. Going forward, we discuss the à la Church approach.

In general, we have an infinite set of type variables $V = \{\tau, \sigma, ...\}$, which provide a type to a λ-term that is a variable. We can create types for the abstractions by combining the type variables in V with the $\rightarrow$ connector. The type $\tau \rightarrow \sigma$ is the type of the λ-abstractions that takes in an τ and with reduction would result in a λ-term of type σ. An example of a λ-abstraction with this type is $\lambda x : \tau.y\ \tau \rightarrow \sigma$. Figure 2.5 illustrates how we inductively builds up $\lambda^{\rightarrow}$-calculus' type.

The syntax of $\lambda^{\rightarrow}$-calculus is almost identical to λ-calculus' syntax. However, in the abstraction, we state the type of the bound variables in the λ binding. Figure 2.6 illustrates $\lambda^{\rightarrow}$-calculus' syntax in BNF, similar to how Figure 2.1 depicts λ-calculus' syntax.

$T :=$	σ	where $\sigma \in V$ (V is the set of variable types)
$\mid$	$M \rightarrow N$	where M and N are valid $\lambda^{\rightarrow}$-calculus types

Figure 2.5: Building up types in simply typed lambda calculs in BNF.

$$\begin{aligned} e := \quad & x \\ | \quad & \lambda x : \tau . M \\ | \quad & MN \end{aligned}$$

Figure 2.6: Simply Typed Lambda Calculus syntax in BNF.

2.2.1 Typing rules

In this section, we present the derivation typing rules as presented by Nederpelt and Herman [31]. However, we first define what a statement, declaration, context and judgment are, as described by Nederpelt and Herman [31]:

- A *statement* is on the form $M : \tau$, where τ is a valid $\lambda^{\rightarrow}$-calculus type and M is a valid λ-calculus term. Here τ is the *type* and M the *subject*.
- A *declaration* is a statement where the subject is a variable.
- A *context* is a set of declarartions with different subjects.
- $\Gamma \vdash M : \tau$ is a *judgement*, with Γ as a context and $M : \tau$ as a statment.

The following three derivation typing rules can be used for $\lambda^{\rightarrow}$-calculus:

(variable) $\Gamma \vdash x : \sigma \; if \; x : \sigma \in \Gamma$

(application) $$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma, \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

(abstraction) $$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau . M : \tau \rightarrow \sigma}$$

2.3 Evaluation strategies

Evaluation strategies are a collection of strategies that decides when a programming language should evaluate an expression. An evaluation strategy chooses, among other things, if the language should evaluate expressions sent into a function before executing the function or send in these expressions and evaluate them at a later stage.

Evaluation strategy has two main strategies *eager evaluation*, also called strict evaluation or applicative-order, and *non-strict evaluation*, also called normal-order. To explain shortly, eager evaluation evaluates the expression as soon as it is bound, while the non-strict evaluation can evaluate an expression when desired. One form of non-strict evaluation is *call by name*, which evaluates an expression when the value is needed. Hence, eager evaluation evaluates the arguments, if they are functions, before substituting them in the function body. On the other hand, non-strict evaluation with a call by name approach substitutes the function body with the expressions, thus, replacing a variable with an expression, not the evaluated value of the expression [1]. Therefore, non-strict evaluation can handle streams, as streams may be infinite and non-strict evaluation only evaluates the parts of the stream it needs. As a consequence of eager evaluation evaluating the arguments before the substitution, we know that an expression is only evaluated once, as it substitutes the evaluated value. Non-strict evaluation with a call by name approach, however, can evaluate the same expression several times, as an expression may replace a variable in more than one place in the substitution.

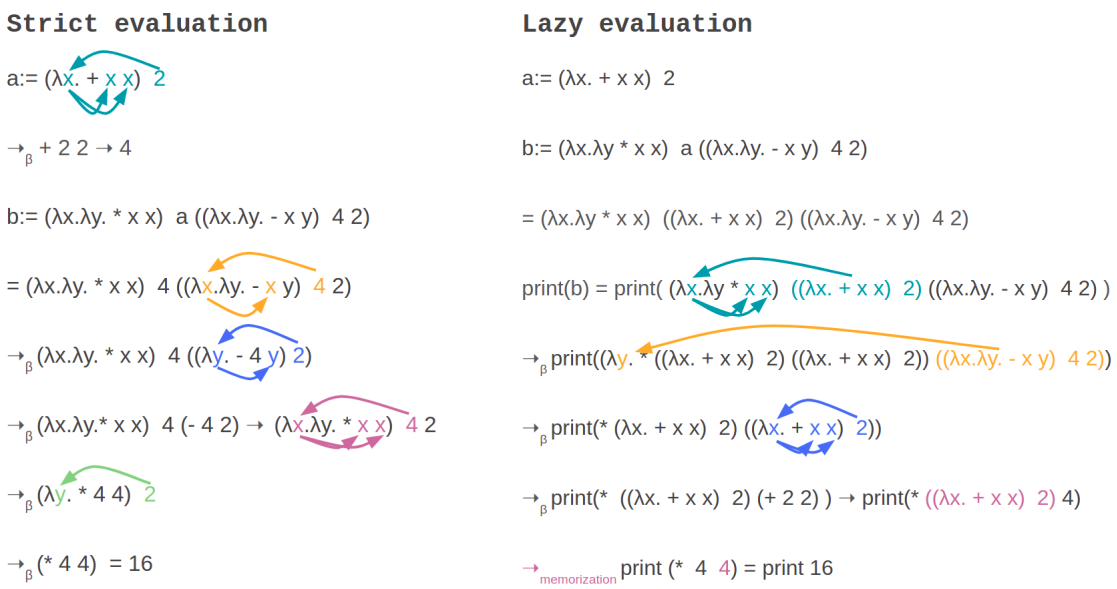

*Figure 2.7: Shows the difference between strict evaluation and lazy evaluation. Here we assume that print is a term which prints the argument to the terminal, and that +, -, and * are operators that work on integers.*

Lazy evaluation is another non-strict evaluation strategy that evaluates the expressions when needed, similar to calling by name. However, lazy evaluation solves the aforementioned problem with the same expression being evaluated several times by including memorisation. Memorisation is implemented by using a look-up table where the evaluated value of a function for some given arguments is stored. When lazy evaluation evaluates an expression, it first checks if the expression already exists in the look-up table. If the value exists, lazy evaluation retrieves the value from the table. However, if the value is not present in the look-up table, lazy evaluation evaluates the function to a value and appends this value to the table. Consequently, lazy evaluation only evaluates an expression when needed and only once. Lazy evaluation is only possible to apply as an evaluation strategy on a purely functional language. In short, a purely functional language is a language that does not allow state changes. Section 2.4 further elaborates on pure functions and purely functional programming languages.

Figure 2.7 is an example of strict evaluation and lazy evaluation with β-reduction. Note the following significant differences between these two approaches in this example:

- Strict evaluation evaluates an expression (terms) as soon as it is bound. In contrast, lazy evaluation waits until the expression is used.
- Strict evaluation evaluates the expressions (terms) before sending them into the term, while lazy evaluation takes in the whole expression.
- On the last two lines in Figure 2.7, lazy evaluation uses memorisation to retrieve the return value rather than evaluating it.

2.4 Functional programming and functional programming principles

Functional programming is a declarative programming paradigm, similar to how object-oriented programming is an imperative programming paradigm. The name of functional programming stems from that a functional program only consists of function calls and arguments [17]. One of the fundamental inspirations for functional programming is Lambda calculus, and many consider λ-calculus as the first functional programming language [16]. Examples of functional programming languages are Lisp[1], Haskel, and Elm. Additionally, several imperative languages have elements that support a functional programming style, such as Java's streams and `Supplier` interface[2]. In this section, we introduce some of the essential functional programming principles.

In Section 2.1, we introduced that λ-calculus regards its functions or applications as first-class values. When a value is a first-class value, we can use the value as an argument, and it can be a return value. Functions that take in functions as arguments or return them are considered *higher-order functions* [16]. Map and filter are two well-known higher-order functions. Map takes in a list and a function that takes in one argument and returns a list where this function is applied on every element in the argument list. Moreover, filter takes in a list and a function that takes in an element and returns a boolean. The filter function returns a subset of the argument list, which only contains the elements of the argument list that, when applied to the function returns true.

Moreover, several functional programming languages are *pure programming languages*, such as Haskell and Elm. A functional programming language that is pure is a *purely functional programming language*. For a programing language to be pure, all of the functions produced in the language must be pure. A *pure function* is a function without any side effects, meaning that a pure function always produces the same output for the same set of inputs. Thus, we can utilise memoisation on pure programming language. On the contrary, using memoisation on a impure language is unreliable because a function retrieving the same set of inputs may yield different outputs.

Several functional programming languages are typed, such as Haskell and Elm. Hence, resembling $\lambda^{\rightarrow}$-calculus. Both Haskell and Elm are strong and statically typed. A *strongly typed programming language* is a programming language where every value has a type. Notably, a strongly typed programming language does not explicitly need to state a value's type. However, a *statically typed language* requires that we explicitly state the type of every variable and parameter. Java is an example of a statically typed language. Regarding functional programming languages, we can think of statically typed languages to follow typing à la Church since we need to state the parameters' types explicitly.

[1] In truth, Lisp is a family of functional programming languages. Common Lisp and Schema are examples of lisp languages.

[2] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Supplier.html`

Chapter 3

Semantic Web & OTTR

The introduction, Chapter 1, introduced Tim Bernese-Lee's vision of the Semantic Web. The vision is to give the data on the Web a formal description and well-defined meaning to enable better cooperation between computers and humans [4]. A set of standards set by the World Wide Web Consortium (W3C) was made to achieve this vision, namely the semantic technologies. Figure 3.1 depicts some of the essential semantic technologies in a stack, which we later refer to as the W3C stack. In this chapter, we introduce RDF [9] and SPARQL [42]. In addition to the W3C recommendation SHACL [26][1]. Moreover, Chapter 1 shortly introduced OTTR and how OTTR can be beneficial for encapsulating a model or ontology. Hence, removing possible repetition of producing RDF graphs. Additionally, Chapter 1 introduced the OTTR's acclaimed benefits. In this chapter, we discuss OTTR's concepts and semantics.

3.1 RDF

RDF [9] stands for *Resource Description Framework* and is a framework for formally describing structured information [13, p. 19]. RDF is the general technolog used in the semantic web to represent information in a web resource, which describes a collection of triples. As the name triple insinuates, a triple consists of three elements, or more formally, three resources: *subject*, *predicate*, and *object*, respectively [9]. The predicate describes the relationship that the subject has to the object. Usually, one can define a collection of triples as a graph where the subjects and objects are nodes, and the predicates are a directed edge from a subject to an object [9]. However, RDF can represent a structure that we ordinarily would not describe as a graph because there are no restrictions in RDF for a resource only being a subject/object or a predicate. Since it is possible to have a resource in the graph representing a node and an edge. An example of a triple is `Sebastian hasFather Thommas`, where `Sebastian` is the object, `hasFather` is the predicate, and `Thommas` is the object. RDF have numerous seralisation formats; we use Turtle [6] in this thesis.

To distinguish between the different resources in an RDF graph, we need to identify the different resources uniquely. Therefore, RDF uses *Uniform Resource Identifier* (URI), where every URI represent a unique resource [13, pp. 21-22]. Therefor our previous triple,

[1] This chapter introduces an overview of these technologies needed to comprehend this thesis. For complete descriptions and formal definitions, see the respective specifications.

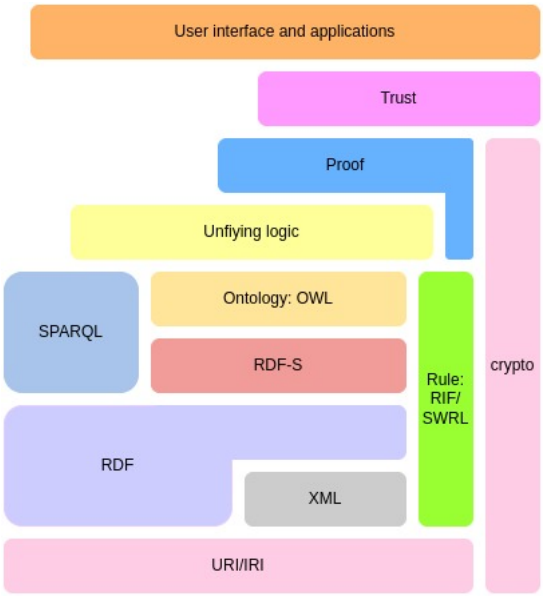

Figure 3.1: What we refer to as the W3C stack or the semantic web stack, containg some of the essential semantic technologies [46].

`Sebastian hasFather Thommas`, needs URIs to uniquely identify the resources; resulting in the following triple with valid URIs `<http://example.org/person/Sebastian><http://example.org/relation/hasFather><http://example.org/person/Thommas>.`. Writing out these absolute URIs may be a time-consuming process. Turtle serialisation solves this issue by making it possible to create prefixes stated at the start of an RDF document. We state these prefixes with a prefix label and a vocabulary URI[2]. We can replace the absolute URIs with their prefix name, a combination of the prefix label and local name separated with a colon (:). For instance, with the prefix `@PREFIX ex-p: <http://example.org/person/>.`, `ex-p:Sebastian` is the prefix name for the absolute URI `<http://example.org/person/Sebastian>`, where `ex-p` is the prefix label referring to the URI `http://example.org/person/` and `Sebastian` is the local name. The examples in the rest of this chapter use the following prefixes:

```
@prefix ex-p: <http://example.org/person/> .
@prefix ex-r: <http://example.org/relation/> .
@prefix ex-t: <http://example.org/template/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema/#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ottr: <http://ns.ottr.xyz/0.4/> .
@prefix o-rdf: <http://tpl.ottr.xyz/rdf/0.1/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
```

Moreover, we may know something about something without knowing what that something is. For example, we may know that Sebastian has a father without knowing who the father is. RDF uses *blank nodes* to model that we know something about a resource without knowing the URI of the resource. We represent Sebastian's father with a blank node

[2]The part of URI that is reused for several resources in our document.

in the case above. Note that a blank node only can occur in a subject or object position, not in a predicate position. To model that Sebastian has a father we can either write `ex-p:Sebastian ex-r:hasFather _:b .` or `ex-p:Sebastian ex-r:hasFather [ ] .` , as Turtle offers two different syntaxes for blank nodes: `_:<some variable name>` and square brackets (`[ ]`). Furthermore, we can express information we know about blank nodes in RDF due to blank nodes being valid subjects. Hence, we can express that Sebastian's father has a father who is `ex-p:Roger`. One way to model the previous statement in RDF is the following set of triples:

Example 3.1.1.

```
ex-p:Sebastian ex-r:hasFather [ ex-r:hasFather ex-p:Roger] .
```

RDF additionally offers *literals*, such as strings and integers [9] which we can use in the object position of a triple. If we want to express that Sebastian has age 22 we can write `ex-p:Sebastian ex-r:hasAge "22"^^xsd:int.`. In addition, we can add that Sebastian has the name Sebastian in Norweigan and Bastian in English by using *language tags*. Expressing the different names in RDF, results in the triples `ex-p:Sebastian ex-r:hasName Sebastian@no.` and `ex-p:Sebastian ex-r:hasName Bastian@en.`. The subsequent graph contains the triples we have mentioned as far in this section:

Example 3.1.2. *RDF triples describing Sebastian.*

```
ex-p:Sebastian ex-r:hasFather ex-p:Thommas .
ex-p:Sebastian ex-r:hasFather [ ex-r:hasFather ex-p:Roger] .
ex-p:Sebastian ex-r:hasAge "22"^^xsd:int .
ex-p:Sebastian ex-r:hasName "Sebastian"@no .
ex-p:Sebastian ex-r:hasName "Bastian"@en .
```

Furthermore, Turtle offers abbreviations to make an RDF document more compact. Firstly, we abbreviate triples with the same subject and predicate but different objects by writing the subject and predicate once and separating the objects with commas (,). Secondly, Turtle also allows us to abbreviate triples with the same subject by writing a semicolon (;) after the object and then continuing with the next predicate[3]. The following graph is equivalent to Example 3.1.2; however, it contains the mentioned abbreviations.

Example 3.1.3. *Same rdf triples as in Example 3.1.2, but with abbreviations.*

```
ex-p:Sebastian ex-r:hasFather ex-p:Thommas, [ ex-r:hasFather ex-p:Roger] ;
               ex-r:hasAge "22"^^xsd:int ;
               ex-r:hasName "Sebastian"@no, "Bastian"@en .
```

Figure 3.2 is a visualisation of Example 3.1.2 and Example 3.1.3.

[3]The last object still needs to end with a dot (.).

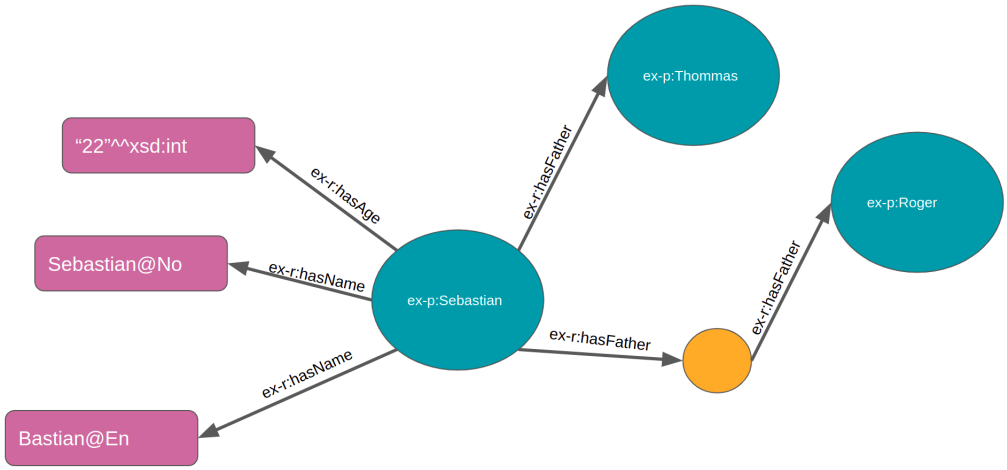

Figure 3.2: The visual graph over Example 3.1.2.

3.1.1 Lists in RDF

In RDF, we have two main ways to represent lists, *containers* and *collections* [13, pp. 58-63]. These two approaches link data together by using blank nodes.

A container has three different types: `rdf:Seq`, represents an order list, `rdf:Bag`, representing an unordered set, and `rdf:Alt`,representing a set of alternatives. We consider a blank node with one of these URIs as its type as a container. These different types of containers do not differ in how we structure the elements in RDF, only how different applications display containers. The container blank node connects to its elements though the predicates `rdf:_1` to `rdf:_n`. Where the first element is in the triple with predicate `rdf:_1`, and the last element, in a list with n elements, is in `rdf:_n`. The following RDF graph shows Sebastians ancestors in an unordered set:[4]:

Example 3.1.4. *An RDF grpah containing Sebastian's ancestorsin a container.*

```
ex-p:Sebastian ex-r:ancestor [a rdf:Bag;
                              rdf:_1 ex-p:Thommas;
                              rdf:_2 ex-p:Roger].
```

Moreover, we can construct lists in RDF using collections. The structure of collections closely resembles a linked list's structure because collections connect blank nodes to each other where each blank node also refers to an element in a collection. Collections use the subsequent predicates and resource to build up a list:

- `rdf:first`: the predicate used between a blank node and its element.
- `rdf:rest`: the predicate used to link a blank node to the next blank node in the sequence of blank nodes.
- `rdf:nil`: the resource in the end of a list.

[4]Note that we have used the abbreviation `a` for `rdf:type`.

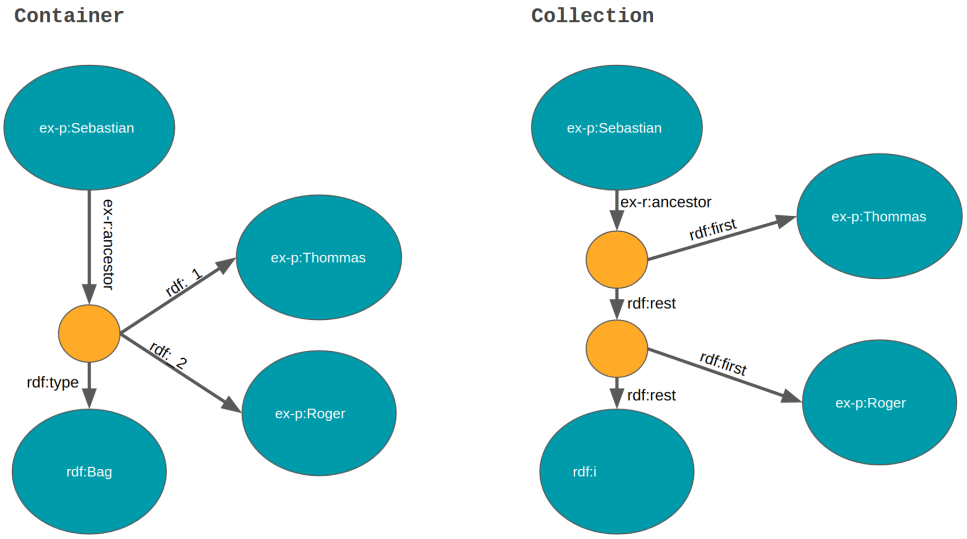

Figure 3.3: Shows the structural differences between a container and a collection.

The main difference between a container and a collection is that a collection is closed, meaning that it is impossible to add new elements after creating a collection. Collections being closed are due to the `rdf:nil` at the end of the collection. On the contrary, it is possible to add new elements to a container, as long as we have the reference the container's blank node. Figure 3.3 visualises the differences between containers and collections.

In Turtle, we can use brackets (`()`) as an abbreviation for writing collections, where we place the elements inside the brackets . Generally, the syntax of a collection looks like this `(element_1 element_2 ... element_n)`. Note that in this thesis, RDF lists refer to collections. An RDF graph containing Sebastian's ancestors using a collection ends up looking like this:

Example 3.1.5. *An RDF grpah containing Sebastian's ancestorsin a collection.*

```
ex-p:Sebastian ex-r:ancestor (ex-p:Thommas ex-p:Roger).
```

3.2 SPARQL

SPARQL [36] enables querying over RDF graphs. SPARQL has several similarities to SQL for Relation Databases, such as the SELECT and WHERE clause. The WHERE-clause is a graph pattern i.e. a set of RDF-tripels that SPARQL searches for in an RDF graph. A graph pattern can contain variables denoted as `?variableName`. SELECT lets us select which variables we want to obtain when executing a query and returns their bindings—applying SELECT * returns all the variables used in the WHERE clause.

As an example, we use the same ontology about families introduced in Section 3.1. Additionally, the graph now also contain the relation `ex-r:lifeStage`, a relation from a person to the person's life stage, e.g. child. All the examples in this section query over the subsequent graph:

```
_:person1 ex-r:hasAge 17;
    ex-r:hasName "Sofie";
    ex-r:hasLifeStage ex-p:Teenager.

_:person2 ex-r:hasAge 16;
    ex-r:hasName "Harry";
    ex-r:hasFather ex-p:Noah;
    ex-r:hasLifeStage ex-p:Teenager.

ex-p:Noah ex-r:hasAge 47;
    ex-r:hasName "Noah";
    ex-r:hasFather ex-p:Fred;
    ex-r:hasLifeStage ex-p:Adult.

ex-p:Fred ex-r:hasName "Fred";
    ex-r:hasLifeStage ex-p:Retired.
```

The following SPARQL query retrieves every person with an age and a name.

Query 3.2.1. *A query retriving every the name and age of every person with a name and age.*

```
SELECT *
WHERE{
    _:person ex-r:hasName ?name;
             ex-r:hasAge ?age.
}
```

The SPARQL query above results in the following table. This table does not contain Fred due to `ex-p:Fred` not having an age, thus not matching the graph pattern in the WHERE clause.

name	age
Sofie	17
Harry	16
Noah	47

In addition, SPARQL offers the UNION pattern, which we can use to combine results from two or more graph patterns. An example of using UNION is to extract the name of every person that is in the retired or teenager life stage.

Query 3.2.2. *A query retriving the name of every person who is teenager or retired.*

```
SELECT ?name
WHERE{
    {
        ?person ex-r:hasName ?name;
                ex-r:hasLifeStage ex-p:Teenager.
```

```
6
7       } UNION {
8           ?person ex-r:hasName ?name;
9                       ex-r:hasLifeStage ex-p:Retired.
10
11      }
12  }
```

name
Sofie
Harry
Fred

Additionally, SPARQL introduces FILTER. FILTER is a clause we can use to filter a result. Furthermore, the OPTIONAL clause allows us to construct an optional graph pattern that may be there, although the pattern is not required.

The subsequent query extracts every person's name and age if the person is under 18 and extracts the father of the person, if the person has a father relationship in the graph; which results in the table bellow.

Query 3.2.3. *A query extracts every person with a name and an age where the age is under 18. The father of this person is also extracted if present.*

```
1  SELECT ?name ?age ?father
2  WHERE{
3      ?person ex-r:hasName ?name;
4              ex-r:hasAge ?age.
5
6      FILTER(?age < 18)
7      OPTIONAL{
8          ?person ex-r:hasFather ?father.
9      }
10 }
```

name	age	father
Sofie	17	
Harry	16	http://example.org/person/Noah

SPARQL 1.1 [42] is an more recent version of SPARQL, introducing, among other things, aggregation, negation, BIND, and property paths. Firstly, aggregation makes it possible to group the result using the clause GROUP BY. Furthermore, the HAVING clause operates over grouped sets, resulting in the possibility to filter on these sets. Both the GROUP BY and HAVING clauses work similar in SPARQL and SQL. Secondly, negation or the NOT EXISTS clause contains a graph pattern of RDF-tripels that should not be present in the graph pattern. Thirdly, SPARQL 1.1 offers BIND, which is a way to

bind a variable to a value. Lastly, property paths open up the opportunity to route specific paths of properties between two resources. There are numerous different types of property paths, such as SequentPath denoted with a slash (/) and AlternativePath denoted with a vertical bar (|). Using the SequentPath, one can define several properties after each other with /. Using the SequentPath to find all the names of all fathers can be written as `?person ex-r:hasFather/ex-r:hasName ?fatherName` in the WHERE clause. The SPARQL 1.1 Query Language document [42] presents a complete table of all the varieties of property paths.

As an example of aggregation, the subsequent SPARQL query results in the number of persons in each life stage if there is more than one person in it.

Query 3.2.4. *A query extracts how many persons there is in each life stage.*

```
SELECT ?stage (COUNT(?person) AS ?persons)
WHERE{
    ?person ex-r:lifeStage ?stage.
}
GROUP BY ?stage
HAVING COUNT(?person) > 1
```

stage	persons
http://example.org/person/Teenager	2

Additionally, the following query uses BIND to calculate the age of a person's father when the person was born.

Query 3.2.5. *A query retriving a person's name, father's name, and father's age when the person was born.*

```
SELECT ?name ?fatherName ?fathersAgeAtBirth
WHERE{
    _:person ex-r:hasName ?name;
             ex-r:hasAge ?age;
             ex-r:hasFather [ex-r:hasName ?fatherName;
                             ex-r:hasAge ?fatherAge].
    BIND(?fatherAge - ?age AS ?fathersAgeAtBirth)
}
```

name	fatherName	fathersAgeAtBirth
Harry	Noah	31

In addition to SELECT, SPARQL offers several other types of queries:

- **CONSTRUCT**: Returns a new RDF graph. The Construct clause contains a graph pattern that applies the variables from the WHERE clause to create the new RDF graph.

- **ASK**: Returns either yes or no. Yes, if the query pattern has a solution, no otherwise.
- **DESCRIBE**: Returns an RDF graph containing data about the resource.
- **DELETE**: Deletes everything that matches the given graph pattern.
- **INSERT**: Works as CONSTRUCT but instead of making a new graph it inserts the pattern inside the INSERT clause into an existing graph.

Additionally, there are various other type of queries as discussed in the document SPARQL 1.1 update language for RDF [34].

3.3 SHACL

The RDF Data Shape Working group had a goal

> ...to produce a language for defining structural constraints on RDF graphs. In the same way that SPARQL made it possible to query RDF data, the product of the RDF Data Shapes WG will enable the definition of graph topologies for interface specification, code development, and data verification [15].

which resulted in, among other things, the Shape Constraint Language (SHACL) [26] . SHACL consists of two main parts: SHACL core and SHACL SPARQL. In this section, we solely focus on SHACL core. SHACL takes in two inputs, an RDF data graph and a shape graph also written in RDF. Shapes are "Conjunctions of constraints that a node must satisfy." [11]. SHACL goes through the data graph and checks if the constraints provided in the shape graph are satisfied, returning a validation report in RDF. The RDF report primarily contains the property `sh:conforms` linking the report to a boolean value. This boolean value is true if the graph does conform to the shapes, otherwise false.

SHACL partitions the shapes into two main types: *node shapes* and *property shapes*. Firstly, a node shape consists of constraints directed at a *focus node*. A focus node or target can be specified in several different ways: targeting all instances of a particular class[5], all resources that are the subject[6] or object[7] of a predicate, or, lastly, directly pointing to a node[8]. Furthermore, a node shape usually contains one or more *property shapes*. A property shape is a constraint directed towards the values that a focus node can reach through a specified property or property path[9]. SHACL has the property `sh:path` to target the property or property path we are after.

Moreover, SHACL makes it possible to restrict the maximum and minimum numbers of distinct nodes often used to constrain how many relations of the path a focus node can have.

[5]Denoted with the property `sh:targetClass`

[6]Denoted with the property `sh:targetSubjectOf`.

[7]Denoted with in a triple with a property `sh:targetObjectOf`.

[8]Denoted with the property `sh:targetNode`.

[9]Property paths in SHACL are a subset of the property path in SPARQL. The complete list of property graphs are available in the SHACL W3C recommendation document [26].

In the following two examples, SHACL performs validations on the following graph:

```
ex-p:Noah a ex-p:Person;
    ex-r:hasAge 47;
    ex-r:hasName "Noah".

ex-p:Fred a ex-p:Person;
    ex-r:hasAge 233352;
    ex-r:hasName "Fred".
```

Example 3.3.1. *A SHACL shape that validates that every person, a resources with the type* `ex-p:Person`*, has minimum one name and exactly one age.*

```
ex-p:PersonShape a sh:NodeShape;
    sh:targetClass ex-p:Person;
    sh:property [sh:path ex-r:hasAge;
                sh:maxCount 1;
                sh:minCount 1;
                sh:name "Age";
                sh:message "Every person needs a age"],
                [sh:path ex-r:hasName;
                sh:minCount 1].
```

Resulting in the following validation report:

```
[] a sh:ValidationReport;
    sh:conforms true .
```

Furthermore, SHACL contains constraint components that define a set of values a node can have. These components can, among other things, specify that the node needs to be a blank node, an IRI, have a particular value, be an instance of a class, or be a literal. We can also specify what kind of data type the literal must be. Additionally, SHACL provides us with several built-in constraint components for the different data types, such as `sh:minInclusive` and `sh:maxInclusive` for numbers and `sh:pattern` and `sh:maxLength` for strings.

Example 3.3.2. *A SHACL shape validates that every person has exactly one age with a positive integer no bigger than 130. Additionally, SHACL validates that there is at least a name that is a string.*

```
ex-p:PersonShape a sh:NodeShape;
    sh:targetClass ex-p:Person;
    sh:property [sh:path ex-r:hasAge;
                sh:maxCount 1;
                sh:minCount 1;
                sh:name "Age";
                sh:maxInclusive 130;
                sh:minInclusive 0;
                sh:dataType xsd:integer;
                sh:message """Every person needs a age that
```

```
                    is between 0 and 130"""],
            [sh:path ex-r:hasName;
            sh:dataType xsd:string;
            sh:minCount 1].
```

Resulting in the following validation report:[10]

```
[] a sh:ValidationReport;
    sh:conforms false;
    sh:result [
        a sh:ValidationResult ;
        sh:resultSeverity sh:Violation ;
        sh:sourceConstraintComponent sh:MaxInclusiveConstraintComponent ;
        sh:focusNode ex-p:Fred ;
        sh:value 233352 ;
        sh:resultPath ex-r:hasAge ;
        sh:resultMessage """Every person needs a age that
                         is between 0 and 130""".
    ] .
```

This validation report yields validation error in Fred's age with the value 233352. The error stems from 233352 not being smaller or equal to 130.

3.4 OTTR

Reasonable Ontology Templates (OTTR) is a language representing ontology modelling patterns as parameterised ontologies, making it possible to produce user-defined abstractions to recurring modelling patterns [44]. OTTR has two central constructors, *templates* and *instances* [45]. Instances are the use of a template, similar to how a function call is the use of a function. Instances contain the name of the template it uses and a set of arguments corresponding to the set of parameters in the affiliated template [45]. Moreover, OTTR templates consist of a head and a body, where the head specifies a template's name and parameters, whereas the body contains the parameterised ontology pattern [44]. The parameters in a template head can be typed with one of the types described in the rOTTR specification [25] of OTTR. If we do not explicitly state the type of a parameter, OTTR interprets the parameter type as `Top`[11]. Hence, OTTR is statically typed, which we further elaborate on in Section 3.4.2.

Additionally to stating the parameter type, we can state that a parameter is optional and non-blank. An *optional* parameter, denoted with a question mark (?), allows a corresponding argument to be none[12]. By default, a parameter is mandatory, meaning that the corresponding value cannot be none. If a parameter is mandatory and a corresponding argument is none, OTTR can discard the instance containing this argument. We discuss when OTTR discards instances more in depth in Section 3.4.4 [44]. On the other hand,

[10] Note that we can make a personalised `sh:resultMessage` with the use of `sh:message`.

[11] `Top` is a supertype for all other types in OTTR, denoted with the IRI `rdfs:Resource`.

[12] None, denoted with `ottr:none`, is a value in the OTTR framework to represent a missing or no value [45], similar to, for example, `null` in Java.

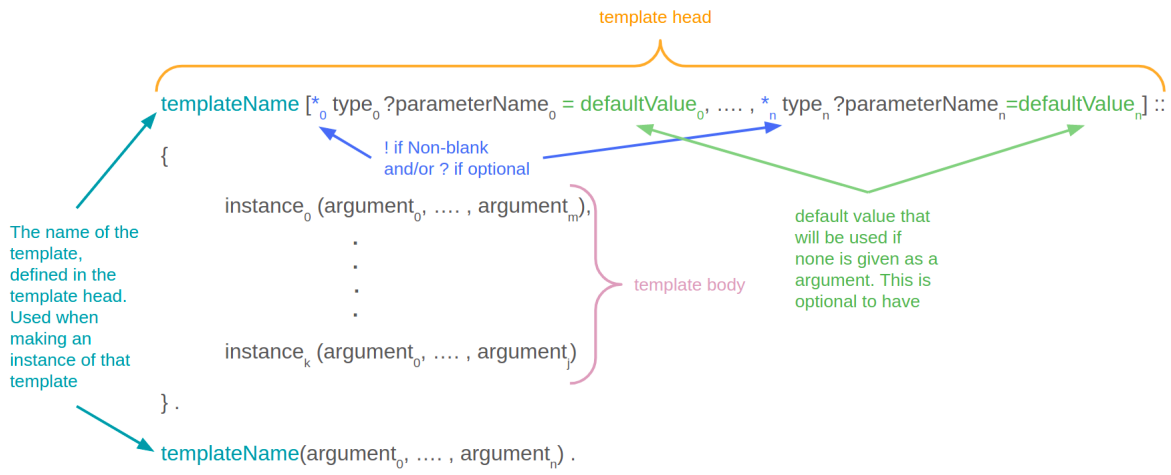

Figure 3.4: A genralisation showing the syntax of stOTTR.

a *non-blank* parameter, denoted with an exclamation mark (!), requires that the corresponding argument not is a blank node [45]. Furthermore, a parameter can also specify a *default value*, a constant. OTTR uses the default value if a corresponding argument is none.

The template body contains instances that refer to templates and base templates. A *base templates* is a particular type of template that does not contain a body and often represents an abstraction in an underlying language. Since OTTR's underlying language is RDF, one critical base template is `ottr:Triple`, representing a single RDF triple [45]. `ottr:Triple` takes in three arguments: a subject, predicate and object, respectively. OTTR expands the instances by substituting its affiliated template's body with its argument. OTTR performs this substitution until it only contains a set of instances on base templates [44], resulting in a RDF graph. Thus, a template body contains the parameterised ontology pattern.

OTTR has two serialisations describing the templates and instances: stOTTR [20] and wOTTR [24]. stOTTER is custom serialisation of OTTR, made to be compact and easy to ready for humans [45]. wOTTR, on the other hand, is a serialisation written in RDF, specified by an OWL ontology and a grammar set by SHACL [45]. In addition to these two serialisations, OTTR also provides two solutions for making instances from structured data sources; bOTTR and tabOTTR. tabOTTR [23] is a markup language that create instances from tabular data files, and bOTTR [22] makes mappings over several queryable sources [45]. Figure 3.4 shows a generalisation of OTTR written in the seralisation stOTTR.

Moreover, OTTR provides a list term; for instance, `(1,2,3)` is the list term containing the integers 1,2 and 3. OTTR offers list expansion to create new instances based on the value in a list. OTTR creates the new instances based on the arguments marked with a list expansion, `++` in front of the argument in stOTTR, and a mode. OTTR treats arguments without a list expansion as a list with one element. Note that the different list expansions behave the same if we only mark one argument with the list expansion [44].

Example 3.4.1. *A OTTR template modelling a person with the same properties as in Section 3.1. The result of expanding the instances on the bottom would be the same as shown in Example 3.1.5.*

```
ex-t:Person [
        ottr:IRI ?person,
        xsd:integer ?age,
        ? List<ottr:IRI> ?fathers,
        ? List<ottr:IRI> ?mothers,
        ? List<ottr:IRI> ?ancestors,
        List<xsd:String> ?names
    ] :: {
        cross | ottr:Triple(?person, ex-r:hasFather, ++?fathers),
        cross | ottr:Triple(?person, ex-r:hasMother, ++?mothers),
        ottr:Triple(?person, ex-r:hasAge, ?age),
        cross | ottr:Triple(?person, ex-r:hasName, ++?names),
        ottr:Triple(?person, ex-r:ancestor, ?ancestors)
}.

ottr:Triple(_:b, ex-r:hasFather, ex-p:Roger) .
ex-t:Person(ex-p:Sebastian, 22 , (ex-p:Thommas, _:b), none,
(ex-p:Thommas, ex-p:Roger), ("Sebastian"@no, "Bastian"@en)).
```

The three different expansion modes work as following [45]:

- `cross`: gives one instance per element in the cross-product.
- `zipMin`: makes one instance per element in the smallest list, making n instances, where n is the length of the smallest list, and combines elements on the same index in the marked lists.
- `zipMax`: almost the same as `zipMin`, but instead of choosing the smallest list, `zipMax` makes one instance for every element in the largest list. OTTR appends none at the end of the smaller marked lists until they are the same size as the largest list.

3.4.1 Terms

Terms in OTTR are the set of constants and variables. A list term is an order collection of terms. We can use the constant `nil` to denote an empty list. In addition to the list term and the constant `nil`, every RDF term, such as IRIs, blank nodes, and literals, are valid OTTR terms. The variable refers to a template's parameters' variables [21] .

3.4.2 Types in OTTR

OTTRs type system has three different types of types: basic types, LUB-type and list types. OTTR arranges its types in a subtype relationship that is transitive and reflexive. The opposite of a subtype is a supertype. As previously mentioned, `Top` is the supertype of

all types. On the other hand, we have the type `Bot`, a subtype of all other types. Moreover, most basic types are common types taken from RDF, RDFS and XSD standards, such as `xsd:integer` and `owl:Class`. However, the OTTR type system also contains basic types that are OTTR specific, for instance, the type `ottr:IRI`.

Furthermore, OTTR offers three complex and OTTR specific types: LUB-type, none empty list type, and list type. Firstly, the LUB-type, denoted with `ottr:Lub`, stands for least upper bound. There exists a LUB type for every basic type `P`, `LUB<P>`, such that `LUB<P>` is a subtype of `P` and that `LUB<P>` is compatible with all supertypes and subtypes of `P`. Moreover, the none empty list type, `NEList<>` denoted with `ottr:NEList`, and the list type, `List<>` denoted with `rdf:List`, are the types of the list terms. A `NEList` must contain at least one element, while a `List` may be empty. The OTTR type system assumes that for each type `P`, in the set of types, there exists a type `NEList<P>` and `List<P>` [45].

OTTR performs validations on the templates and instances. Among these validations, OTTR validates *compatible typing* and *consistent typing*. In short, every type `P` that is a subtype of `Q` is also compatible with `Q`. However, as the previous paragraph states, `LUB<P>` is not only compatible with all of `P` supertypes but also with the subtypes of `P` [21]. OTTR validates that the types of the arguments in an instance are compatible with its corresponding argument. Moreover, consistent typing consists of two parts: type compatibility and *inferred typing*. Inferred typing refers to that a term `v` has inferred type `P` if the term is used in argument `a` and `a`'s corresponding parameter's type is `P`[13]. OTTR considers the term `v` to be consistently typed if there exists a type `P`[14], such that `P` is a subtype of all inferred types of `v` and that `v`'s type is compatible with `P` [21]. consistent typing is especially important for blank nodes, which have the type `Bot`, thus, being compatible with all other types; since consistent typing ensures that a blank node is not placed in several arguments referring to parameters that are not compatible with each other. For instance, a blank node `b` cannot be used as an argument with a corresponding parameter type `xsd:integer` and another argument with corresponding parameter type `xsd:string`, as there does not exist a type `P` that is a subtype of both `xsd:integer` and `xsd:string`.

3.4.3 Template library and template dataset

A template library in OTTR is a set of template signatures. A template signature contains the name of the template and its list of parameters. Thus, we consider both templates and base templates as a template signatures. Furthermore, a template dataset consists of template ground instances, i.e. instances where all the arguments are constants, and a template library. The following definitions regard template libraries:

Definition 3.4.1. *A template T* ***directly depends*** *on a template signature S, iff the template body of T contains an instance of S. Thus,* ***directly depends*** *is a relation between templates and templates signatures [21].*

[13] The rule for inferred types are slightly different if a has a list expander, see the mOTTR specification [21].

[14] Unequal to `Bot`.

Definition 3.4.2. *For a template library to be* ***acyclic*** *the directly depends relation on the library needs to be acyclic [21].*

3.4.4 Expansion of OTTR instances

OTTR expands instances resulting in an RDF graph; hence the templates can encapsulate a model over a domain. As previously mentioned, OTTR recursively expands the instances by substituting templates until we reach a set of only base templates without list expanders. However, different concepts in OTTR affect the semantics of expansion in OTTR. Firstly, OTTR discards an instance that contains none as an argument in an instance where the corresponding parameter is mandatory and does not contain a default value. Discarding an instance in OTTR means to not expand this instance. Secondly, OTTR creates new instances if an instance contains a list expander. How OTTR creates new instances depends on the mode, which we previously discussed in this section.

Chapter 4

Design

As mentioned in Chapter 1, OTTR claims to give various benefits over using plain RDF and OWL, such as *better abstraction* and *Separation of design and content*. However, we propose that it is possible to strengthen the benefits of OTTR by including a *statically and strongly typed purely functional programming language*, namely Frog. Frog makes it possible to construct functions, which we can use by creating function calls in the OTTR templates and instances to manipulate terms. This chapter discusses our design of Frog, a programming language that seamlessly integrates with OTTR.

In Section 4.1, we give an overview of Frog, including a description of functions and functionals. Section 4.2 elaborates on Frog's syntax by presenting Frog's two serialisations: the RDF syntax and the Human Readable Syntax (HRS). Moreover, Section 4.3 discusses the necessary additional type to OTTR's type system, the function type, while Section 4.4 examines why Frog benefits from being a generic language and introduces how generic works in Frog. Furthermore, OTTR performs several types of validations on templates and instances. One of these validates that the arguments used in the instances are compatible with their corresponding template parameter. Expanding OTTR with Frog allows functions and function calls as arguments in instances. Therefore, OTTR must validate that the function calls and functions used arguments are compatible with their corresponding parameter. For the validation of function calls to be correct in OTTR, Frog must validate Frog functions, which we discuss in Section 4.5. Finally, in Section 4.6, we discuss Frog's evaluation strategy and when OTTR should evaluate a function call in instances.

The following prefixes will be used in the examples of this chapter:

```
@prefix ex: <http://example.xyz/ns/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix fn: <http://ns.frog.ottr.xyz/0.1/function/> .
@prefix : <http://ns.frog.ottr.xyz/0.1#> .
@prefix ottr:  <http://ns.ottr.xyz/0.4/> .
@prefix o-rdf: <http://tpl.ottr.xyz/rdf/0.1/>.
```

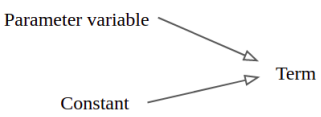

Figure 4.1: Frog terms.

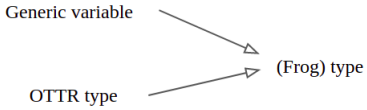

Figure 4.2: Frog types.

4.1 Overview

This section introduces Frog's main constructs, namely *functions* and *function calls*. Both the functions and function calls are terms in both OTTR and Frog, represented with *function call terms* and *function terms*. Moreover, we define correctness defintions on the newly introduced terms and dependencies relation between functions and functions and templates and functions.

4.1.1 Concepts

As previously introduced, Frog consists of functions and function calls; these are the primary constructs in Frog. By creating functions, we can define calculations on a set of terms and perform these calculations by creating function calls on the functions. Placing function calls inside templates' bodies, as arguments in instances, makes it possible to conduct calculations on templates' terms. OTTR replaces the function calls with their evaluated values during the expansion[1]. A function call is a valid term since we use function calls as arguments, and arguments, by definition, are terms. Function calls being terms means that they must have a type, as OTTR needs to validate that the function call terms are compatible typed. A function call's type is the type of the value it evaluates to; since Frog is a statically, a function call's type is the return type of the function it evaluates. Moreover, to be able to use functions as arguments in both OTTR and Frog, we introduce a function term, which is a reference to a function. Similar to the function calls, OTTR requires the function terms to have a type for OTTR to confirm that the terms are compatible typed. As functions are a new construct, similar to how a list is a construct, we introduce a new type for function terms to the OTTR type system, the *Function type*. Section 4.3 goes in-depth on this new Function type.

Frog and OTTR use the same term system, as we argue that using the same term system creates a seamless integration of Frog since no translation of terms between Frog and OTTR is needed. Hence, the newly introduced terms, function term and function call, are similar and valid terms in both OTTR and Frog. As Frog and OTTR have the same term system, Frog uses OTTR's type system to type its terms[2]. However, a valid type in Frog can be OTTR type or a generic variable, as Frog is a generic language. A generic variable in Frog is a placeholder for an OTTR type. Hence, we consider OTTR and Frog to have the same type system even though Frog also considers generic variables as valid types.

[1] Section 4.6.2 discusses the semantics of function call evaluation in OTTR.

[2] We argue that using the same type system is beneficial for the same reason regarding equal term systems.

A Frog function is constructed in two parts, the *function head* and the *function body*. The function body defines what the function does under evaluations and consist of only one element, a function call. On the other hand, the function head specifies the function IRI, parameters, return type and the generic arguments. The function IRI is an IRI to identify the function uniquely; in other words, the function name. Furthermore, the function head consists of an arbitrary number of parameters. As a consequence of Frog being a typed language, Frog needs to know the parameter type: therefore, the parameter consists of a parameter type and a parameter variable. The parameter variable represents a constant, and this variable can be used as arguments in the function body. Additionally, if the parameter type is a Function type, Frog can use the parameter variable as a function in function calls in the function body[3]. Moreover, the parameter type is also needed when conducting validation, as discussed in Section 4.5.

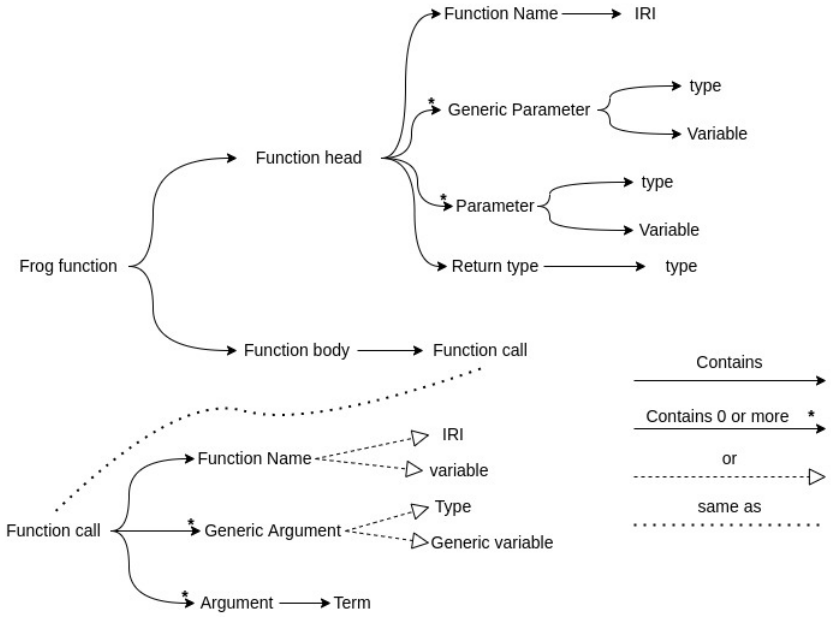

Figure 4.3: The strucure of functions and function calls.

Additionally, the function head specifies a possibly empty list of generic parameters, each consisting of a variable and a type, much like an ordinary parameter. However, in this case, a generic variable represents a type that is a subtype of its belonging type. A generic variable can occur in three places within a function definition:

1. As a parameter type in the function head.
2. As the return type in the function head.
3. As a generic argument in the function calls found in the function body.

Finally, the function head specifies the return type, representing the type of the value that the function returns. The type of the value returned by the function body needs to be a subtype of the return type defined in the function head.

[3]Similarly, if a template's parameter's type is of the function type, then a function call in this template's body can use this parameter variable as its function.

Furthermore, a function call consists of a function name, arguments and generic arguments. Firstly, the function name defines which functions to execute. The function name can either be an IRI referring to a function's name or a parameter variable of the function type, as previously explained. Secondly, the function call arguments can either be constants or parameter variables, i.e. a term, as illustrated by Figure 4.1. Lastly, a function call contains an arbitrary number of generic arguments. A generic argument can either be a generic variable or a type. The generic argument needs to be a subtype of the affiliated function's corresponding generic parameter. Note that both the arguments and the generic arguments may be empty. Figure 4.3 is a visual representation of the function and function call structure.

A function term consists of the function it refers to's IRI and a possibly empty list of generic arguments. The function terms are equivalent to IRI terms when the list of generic arguments is empty. Therefore, the type of the arguments corresponding parameter determines whether OTTR and Frog should interpret such a term as an IRI term or a function term. Note that if the corresponding parameter is `Top`, then OTTR and Frog interpret always interpret the term to ba an IRI.

Moreover, there are two types of functions in Frog: base and Frog functions. The difference between base functions and Frog functions is that base functions do not contain a function body. Instead, base functions contain an IRI, which refers to the task they perform. The base functions are a combination of XPath functions and built-in java functions. On the one hand, the XPath functions work on the terms of the base type, offering functions such as addition of numbers and concatenation of strings. On the other hand, the built-in java functions work on the terms with a complex type, such as removing the first element of a list. A base function is built into the Frog and OTTR system, and we can execute them in the same manner as with Frog functions. Note that a Frog programmer cannot create base functions, only Frog functions. A Frog function defines a way to combine base and Frog functions to perform a specific task. For instance, a function multiplying every number in a list with the number five[4].

4.1.2 Abstract Model

In this section, we introduce the definition of the correspondence between function calls and functions, a function library, and the correctness of the two new terms: the function call term and the function term. Furthermore, OTTR presents several concepts on dependencies between templates. For example, a template directly depends on a template signature if an instance of the template signature exists in the template body of the template [44]. Frog expands these concepts with four new dependencies between functions and functions and template and functions, as defined in the following definition in this section.

Definition 4.1.1. *A function call's* ***affiliated function*** *is the function the function call executes. A function terms* ***affiliated function*** *is the function it refers to.*

[4]We have not defined the syntax of Frog yet. However, the function that multiplies every number in a list with five can be found in Figure 4.21. The function is in the human-readable syntax, which we outline in Section 4.2.3.

Definition 4.1.2. *Let* ***F*** *be a function with* ***n*** *parameters, (P_1, ..., P_n) and* ***FC*** *be a function call with* ***n*** *arguments, (A_1, ..., A_n) with* ***F*** *as its affiliated function. Then for an arbitrary $1 \leq i \leq n$ F A_i's* ***corresponding parameter*** *is P_i.*[5]

Definition 4.1.3. *A* ***function library*** *is a set of Frog functions.*

Definition 4.1.4. *A* ***correct function call*** *is a function call that:*

- *has a name affiliated with a function in the function library or a variable defined in either the template head or the function head that is explicitly typed as a function.*
- *has an arity of arguments equal to the arity of parameters in the affiliated function.*
- *has an arity of generic arguments equal to the arity of generic parameters in the affiliated function.*
- *for every argument the type of the argument is a subtype of the corresponding paramters type.*
- *for every generic argument the generic argument is a subtype of the corresponding generic paramters defined subtype.*

Definition 4.1.5. *A* ***correct function term*** *is a function term that:*

- *has a name affiliated with a function in the function library.*
- *has an arity of arguments equal to the arity of parameters in the affiliated function.*
- *for every generic argument the generic argument is a subtype of the corresponding generic paramters defined subtype.*

Moreover, Frog substitutes the arguments into the affiliated function's body when executing a function call. Thus, the arguments in the function call replace their corresponding parameter in the affiliated function's function body. Similar as with arguments, generic arguments replace their corresponding generic parameter in the function body. Frog performs the substitution recursively until Frog reaches a state only containing constants and base functions. Figure 4.4 illustrates the recursive manner in which Frog substitutes the parameters and generic parameters with the function calls arguments and generic arguments. As mentioned in Section 2.1.3, some λ-terms never reach β-normal form when performing β-reductions, hence being non-terminating. Similarly, a Frog function call may be non-terminating, primarily due to Frog recursive manner.

4.2 Syntax

As introduced in Section 3.4, there are two serialisations of Frog: wOTTR and sTOTTR. Skjæveland et al. describe sTOTTR as the serialisation of OTTR made for easy reading and writing by humans, in addition to being compact [45]. On the other hand, the wOTTR serialisation offers the benefit of leveraging the existing W3C stack, in addition

[5]The same holds for generic arguments and generic parameters, and with function terms.

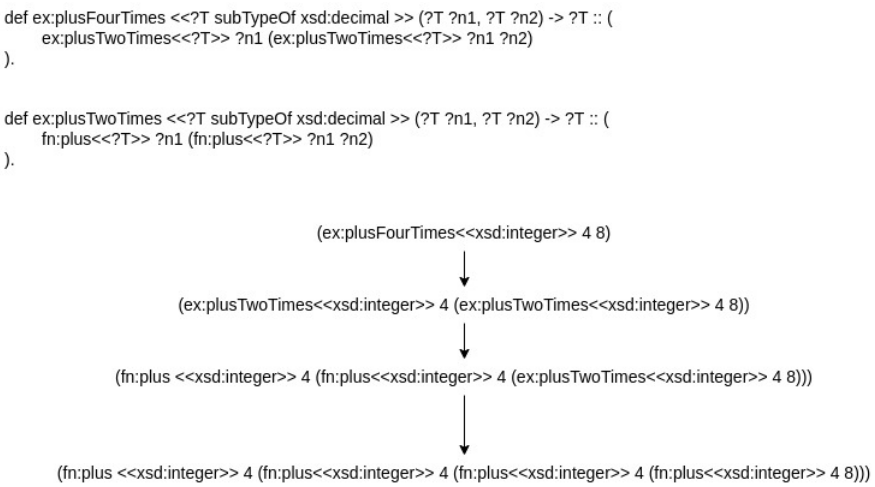

Figure 4.4: An example of how Frog performs the substitution when executing a function call.

to tools for developing, publishing and maintaining templates [44]. This section begins by arguing for the necessity of Frog offering two serialisations.

As mentioned, wOTTR is OTTR's RDF serialisation and the preferred choice for publishing templates. With the addition of Frog, we make it possible to create user-defined functions and apply these functions by creating function calls inside the templates. Thus, it is natural to assume that a template programmer publishes the Frog functions made for a template library together with the templates in a similar format. Consequently, we consider it necessary for Frog to provide a serialisation in RDF. Moreover, OTTR benefits from wOTTR leveraging the existing W3C stack. We argue that to preserve the benefit of leveraging the existing W3C stack, everything integrated with OTTR must also be able to leverage. As a result of the arguments above, we have included an RDF syntax of Frog.

However, only offering an RDF serialisation of Frog proposes a challenge: constructing several functions requires repetitions. This repetition stems from from how RDF structures data in triples, resulting in unnecessary metadata that a human does not need to comprehend. The problem concerning repetition and RDF when creating similar RDF structures is not new and is, in fact, partially removed when utilising OTTR to produce an RDF graph. Hence, a template modelling a Frog function could be a solution to remove tedious repetitions when creating different Frog functions. A template may be beneficial when creating the function head. However, providing a template that models the function body may not be attainable due to the difficulties of creating a template that can model a general function call. However, we did not find an optimal manner of constructing a template encapsulating the function body in the RDF syntax due to the special list structure used to represent function calls. Therefore, we propose a second compact serialisation, similar to sTOTTR, that is compact and easy for a human to read and write.

We present other benefits of a readable serialisation. Firstly, having the human-readable

serialisation does not require the programmer to understand RDF to produce functions that the template can use. Hence, we argue that this serialisation allows more people to create Frog functions. However, this claim is only valid if the syntax is easy to learn for an experienced programmer. Consequently, we have taken inspiration from other well-known programming languages' syntaxes when creating Frog's human-readable serialisation. Secondly, having a human-readable serialisation makes it possible to shorten the function calls, hence shorting the function body, as shown in Figure 4.7 compared to Figure 4.5.

In summary, Frog provides two serialisations, similar to OTTR: one in RDF and one compact and easy to read and write for humans. Thus, each OTTR serialisation has a counterpart in Frog, with similar benefits. The rest of this section introduces Frog's two serialisations.

4.2.1 Similarities in the two syntaxes

Section 4.1.1 mentioned that Frog and OTTR use the same type and term systems, including the newly introduced function call terms, function terms, and the function type. Consequently, we have chosen that syntaxes for terms and types in Frog's two serialisations to be equal to their counterparts in OTTR. Hence, supporting a seamless integration between OTTR and Frog.

A function call has a similar syntax in both serialisations. LISP languages, such as Scheme, have inspired the syntax of the function calls in Frog, implying that the function IRI and the arguments are in the same list. The advantage of using a list structure for function calls is that lists are an established data structure in RDF. However, utilising list has its challenges as Section 4.2.2 addresses.

4.2.2 RDF Syntax

In this section, we informally introduce Frog's RDF syntax. Formally, the RDF serialisation is, as with wOTTR, defined by a set of SHACL shapes and OWL vocabulary. Due to Frog and OTTR having the same terms and types, the SHACL shapes for validating

```
ex:FtoC a :Function;
    :type [:returnType xsd:decimal; :parameterTypes (xsd:decimal)]
    :def (:lambda (_:fahrenheit)
        (:functionCall fn:times (:typeArgs xsd:decimal)
            (:functionCall fn:minus (:typeArgs xsd:decimal) _:fahrenheit 31)
            (:functionCall fn:divide (:typeArgs xsd:decimal) 5 9)
        )
    ).
```

Figure 4.5: An example of a Frog function that converts Fahrenheit to Celcius using the RDF turtle syntax.

terms and types in Frog are equal to those found in wOTTR's specification. Appendix A.1 contains the RDF syntax's SHACL shape and OWL vocabulary.

Figure 4.6 shows a generalisation of the RDF syntax and how the different components of a function and function call relate to each other; see Figure 4.5 for an actual Frog function in the RDF syntax. The following list describes the vocabulary used by Frog's RDF syntax:

- `:Function` : For Frog to interpret an IRI as a function, we need to explicitly state that the type of the IRI is a Function.
- `:type` defines the type of the function, and is a relation between a function IRI and a blank node representing the type.
- `:returnType` defines a function's return type and relates the blank node representing the type to a type.
- `:parameterTypes` defines the types of the parameters and relates the blank node representing a potentially empty list containing the types to the parameters.
- `:typeVars` defines the generic parameters of the function and is a relation between a function IRI and a list containing blank nodes representing the generic parameter.
- `:var` defines the generic variable of a generic parameter and is a relation between a blank node representing the generic parameter and a blank node.
- `:subtypeOf` defines the subtype relationship between the generic argument and is a relation between the blank node representing a generic parameter and a type.
- `:def` is a relation between the function and a list containing `:lambda`, a list of the parameter variables, and the function call, in that specific order.
- `:lambda` is the first element in the list containing a second element that is a parameter list, in addition to a third element that is the function call.
- `:functionCall` defines that Frog should interpret the list as a function call. The `:functionCall` needs to be the first element of the list.
- `:typeArgs` defines that Frog should interpret the list as a list of generic arguments. `:typeArgs` need to be placed at the start of the list. If the list of generic arguments is present, Frog demands the list to be the second element in the function call list.
- `:functionTerm` defines that Frog should interpret the list as a function term. The list must consist of three elements, where the first element is `:functionTerm`, the second element is an IRI referring to a base function or a function in the function library, and a generic argument list.

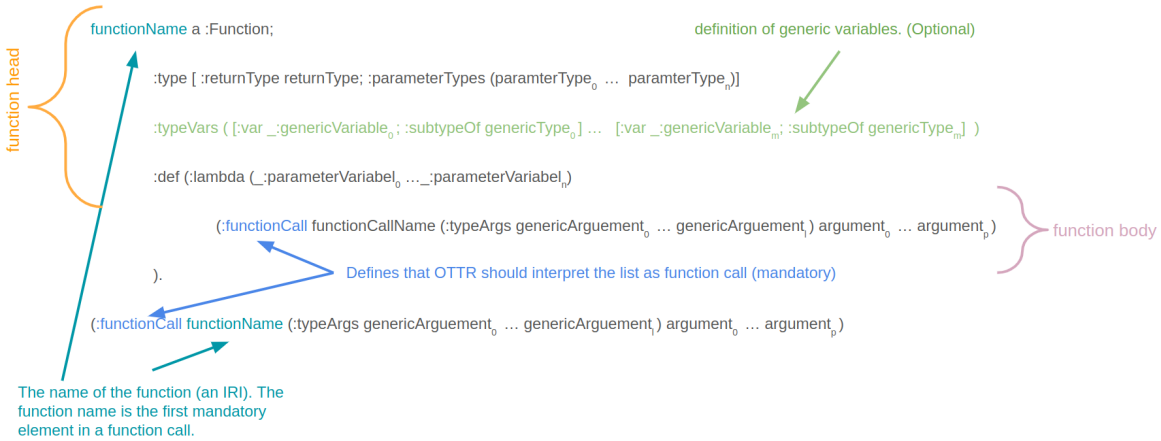

Figure 4.6: A genralisation showing Frog's RDF syntax.

The list problem

As previously presented, the RDF serialisation and the Human Readable syntax use lists as their syntax for function calls. In addition, the RDF syntax expresses generic arguments, as seen in Figure 4.5, and function terms with lists. The benefits of using lists are that lists are established structures in RDF. Additionally, as introduced in Section 3.1.1 and by Figure 3.1.5, the Turtle syntax of RDF offers a compact form for writing lists. Moreover, using lists in both syntaxes make them more similar. We argue that the similarities in the serialisations makes it easier to understand one of the serialisations based on the other.

However, using lists as function calls proposes a problem in the RDF syntax, namely that Frog and OTTR can interpret an RDF list where the first element is an IRI as either a function call or a list. For instance, the list `(fn:plus 2 3)` can have two interpretations. Firstly, `(fn:plus 2 3)` can refer to the list containing the elements `fn:plus`, `2` and `3`. Secondly, `(fn:plus 2 3)` can refer to the function `fn:plus`, which adds together `2` and `3`.

Additionally, as Figure 4.5 illustrates, we have chosen to express the generic arguments and function terms with lists for the same reasons as mentioned regarding function calls: lists being an established ordered compact structure in Turtle. As a result, Frog and OTTR may interpret a list as a literal list, a function call, a function term, or a list of generic arguments.

To solve the issues above, we propose that function calls, function terms, and generic arguments contain a key IRI as the first element such that Frog and OTTR decode the list correctly in their RDF syntaxes. The first element in a function call list is the IRI `:functionCall`, the first element in a function term list is `:functionTerm`, while the first element in a generic argument list is `:typeArgs`. As a result, Frog and OTTR decode `(fn:plus 2 3)` as a list and `(:functionCall fn:plus 2 3)` as a function call. We argue that this explicit solution is less error-prone than an implicit interpretation of the lists. Additionally, we consider that using key IRIs makes it easier to produce more

concrete and correct validation messages.

4.2.3 Human Readable Syntax

We have based Frog's Human Readable syntax (HRS) on well-known programming languages, such as python, java and LISP. Additionally, since Frog is an integrated part of OTTR, we have taken inspiration from the stOTTR's syntax of templates when designing the HRS's syntax of functions. In short, a combination of python, java and OTTR inspires the syntax for the function head and generic arguments. While LISP, on the other hand, inspires the function body.

```
def ex:FtoC(xsd:decimal ?fahrenheit) -> xsd:decimal :: (
    fn:times<<xsd:decimal>>
        (fn:minus<<xsd:decimal>> ?fahrenheit 31)
        (fn:divide<<xsd:decimal>> 5 9)
).
```

Figure 4.7: An example of a Frog function that converts Fahrenheit to Celcius using the Human Readable syntax.

As previously discussed, the HRS inherits the syntax of terms and types from stOTTR. An example of this inheritance is the parameter variables that HRS depicts with an question mark (`?`) followed by the parameter name precisely like stOTTR. However, there are consequences of Frog inheriting the syntax terms and types from stOTTR. Namely, some frequent patterns used to express certain elements in programming languages already exist, such as arrow brackets (`<>`) for defining generic parameters and arguments. In this case, the arrow brackets (`<>`) are already a means to express absolute IRIs, for instance, `<http://ns.frog.ottr.xyz/0.1#functionCall>` which is the same as the abbreviated form of `:functionCall`. OTTR and Frog using arrow brackets (`<>`) to express IRIs are due to them having a subset of Turtle's syntax [45]. As a result, we chose to use double arrow brackets (`<<>>`) to express the list of generic parameters and arguments to make it similar to how Java expresses generics.

Figure 4.8 illustrates a general Frog function in the HRS, while Figure 4.7 shows a real example of a Frog function written in HRS. The following points summarise some of the HRS's vocabulary regarding functions; Appendix A.2 contains a complete and formal definition of the HRS.

- The function's name is an IRI uniquely identifying the function. Similarly to the function name in the RDF syntax and the template name in stOTTR. Before the function name, `def` is used to define the start of the function, as in python.

- The list of the generic variables are defined inside double arrow brackets (`<<>>`) and separated with comma (`,`). To define what type the generic variable is a subtype of, we write the keyword `subtypeOf` between the generic variable and type.

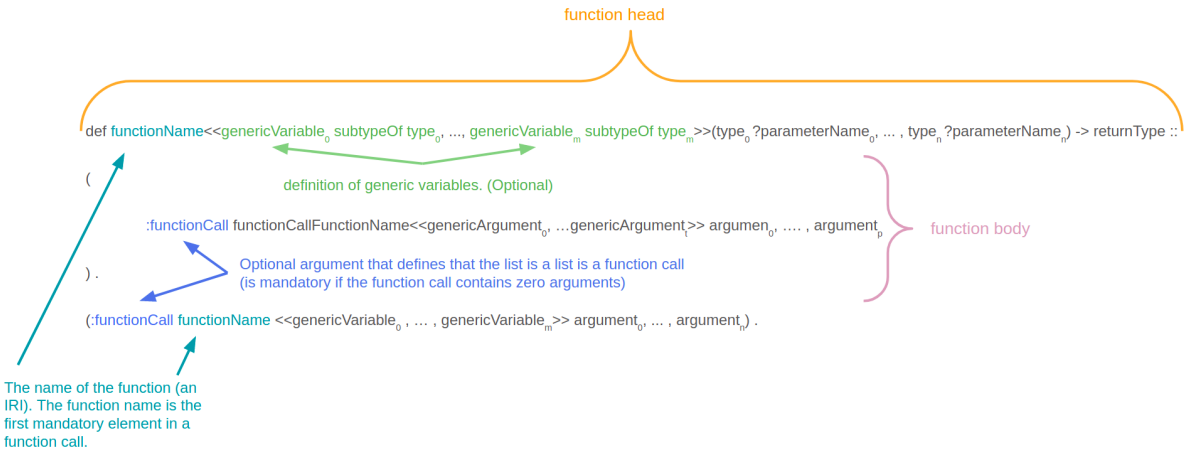

Figure 4.8: A figure illustrating Frog's human readable syntax.

- The list of parameters are defined inside parentheses (`()`) and separated with comma (`,`). The HRS defines the parameter type before the parameter name, similar to Java and OTTR.
- The HRS defines the return type after the list of parameters with an rigth arrow (`->`) followed by the type of the return value.
- Frogs HRS uses a double colon (`::`) as a distinction between the function head and body, similar to how sTOTTR seperates the template head and body.
- The function body is defined as a function call.
- Every function ends with a period (`.`) similar to how templates end with an period (`.`) in stOTTR.

A function call consists of four parts in a specific order: the `:functionCall` IRI[6], the name of the function it is applying, a possibly empty list of generic arguments, and lastly, a possibly empty list of arguments. Note that a function call in the HRS is not utilising stOTTR's means of describing a list by separating the elements with comma (`,`), but rather the turtle syntax, separating the elements with spaces. Consequently, making it easier to separate lists and function calls, compared to the RDF syntax. However, Frog and OTTR interpret a function call with no arguments as both a list and a function call. Therefore, we have designed function calls such that it is mandatory to use the IRI `:functionCall` on a function call with an empty list of arguments. In all other cases, using the `:functionCall` IRI is optional. Moreover, the HRS expresses the list of generic arguments similar to how it expresses generic parameters, with double arrow brackets (`<<>>`) and comma (`,`) for separation.

4.3 Extending the OTTR type system

This section discusses the necessity for a new type, the Function type, in the OTTR type system. Furthermore, we discuss the criteria for the function type and formally define

[6]The `:functionCall` is optional if there are one or more arguments.

```
def ex:higherOrder(Function<xsd:integer, xsd:integer, xsd:integer> ?fn)
    -> xsd:integer :: (
    ?fn 5 6
).
```

Figure 4.9: An example of an higher-order function in Frog, which in this case takes in a function.

```
ex:TWFP[Function<xsd:integer, xsd:integer> ?fn, xsd:integer ?nb] :: {
    ottr:Triple(ex:Test, ex:functionResult (?fn ?nb))
} .
```

Figure 4.10: An example of a OTTR template where one of the parameters are a function.

the type and the subtype relationship between function types. Finally, we address the syntax of the function type and the various serialisations of OTTR and Frog.

The primary reason for needing a function type is for OTTR and Frog to be able to perform correct validations. OTTR and Frog validate that an argument in an instance or a function call is compatible with the type of the corresponding parameter. The mOTTR specification defines every term as a valid argument [21]. Consequently, the newly introduced terms must have a type. As Section 4.1.2 mentioned, a function call evaluates to a term that is not a function call; thus, the type of a function call is the return type of its affiliated function. Therefore, the function call term does not require a new type.

However, the OTTR type system does not contain a natural type for a function term. As illustrated by Figures 4.9 and 4.10, both templates and functions can contain function calls where the function is a parameter variable. OTTR and Frog have to validate that the function call is correct, which we elaborate on in Section 4.5. A criteria for a function call to be correct is that the name either is a parameter variable of the function type, an IRI refering to function in the function library or an IRI refering to a base function. Consequently, OTTR and Frog need to know explicitly that the parameter variable is in fact a function. Moreover, a function call is only valid if every argument is a subtype of the corresponding parameter's type; as a result, OTTR and Frog need to know the type of the parameter variable's parameter types. Finally, as OTTR and Frog validate the correctness of the arguments, they need to know the type of the function call. As previously stated, the type of the function call depends on the return type of its affiliated function. Thus, OTTR and Frog need to know the return type of a variable used as a function in a function call.

Another argument for introducing a function type is that an IRI can both represent an IRI and a function without any generic arguments. Having a function type makes it clear

for OTTR and Frog whether they should interpret a term as an IRI or a function[7].

The previous paragraphs express the need for a new type representing the function terms, where we can retrieve information regarding the types of the parameters and return value. Consequently, we have constructed a function type that contains n type arguments, where the type arguments 0 to $n-1$ represent the parameter types and argument n represents the return type. A valid function term contains at least one type argument since every function has a return type. Definition 4.3.1 formally defines a Function term, while Definition 4.3.2 defines the function type's subtype relationship.

Definition 4.3.1. *The* ***Function*** *type takes in n other types as type arguments,* ***Function***$<T_1,...,T_n>$*, where* $T_1,...,T_{n-1}$ *is the type of the function parameters, and* T_n *is the functions return type.* $n > 0$*, as every function has a return type. We assume that there is one type* ***Function***$< P_1, ..., P_n >$ *per combination of* $P_0, ..., P_n$*, where* $P_0, ..., P_n$ *can be any other type.*

Definition 4.3.2. *The function type* ***Function***$< P_1, ..., P_n >$ *is a* ***subtype*** *of the type* T *if* T *is* ***TOP*** *or if* T *is the function type* ***Function***$< T_1, ..., T_m >$ *where* $n = m$ *and for any* $1 \leq i \leq n-1$ T_i *is supertype of* P_i *and* P_n *is a subtype of* T_m

4.3.1 Syntax of the function type

The syntax for the LUB and lists from OTTR types has inspired the syntax for the function type[8]. wOTTR expresses the list and LUB types through RDF lists of IRIs. Where `rdf:List` and `ottr:NEList` denotes the `List` and `NEList`[9] type, respectively, and `ottr:LUB` denotes the LUB type [25]. For instance, `(rdf:List ottr:NEList xsd:string)` is the type of a `List` containing `NEList`, which further contains strings. Moreover, `(ottr:LUB xsd:integer)` denotes the `LUB`-type over the integer type. Similar to the list and LUB types, the RDF serialisation of OTTR and Frog also construct the function type from RDF lists and IRI, where the IRI `:Function` denotes a function. However, one function type may contain several type arguments; thus, we need to be able to separate the type arguments. The syntax of Frog function type in the RDF serialisation permits nested lists, in contrast to the LUB and lists types. `(:Function (rdf:List xsd:integer)(:Function xsd:integer xsd:string)xsd:string)` is an example of a function type in the RDF serialisations. This function type is the type of every function taking in a list of integers and a function that takes in an integer and returns a string, and returns a string.

Moreover, stOTTR depicts the LUB and list type with a keyword, `LUB` for the LUB-type and `List` and `NEList` for the two list types, and with the type argument inside arrow brackets (`<>`). stOTTR expresses the examples given in the previous paragraph as `List <NEList<xsd:string>>` for the list containing lists of strings and `LUB<xsd:integer>`

[7]Note that if a term that can be interpreted as a function term and an IRI term with a corresponding parameter that has the type `TOP`, then the argument is interpreted as an IRI term.

[8]As mentioned in Section 4.2, Appendix A stores the formal definitions of the syntax, including the Function type, for both serialisations.

[9]Non empty list.

for the LUB-type of `xsd:integer`. Similarly, the function type has a keyword, namely `Function`, and stores the type arguments in `<>`. However, as previously mentioned, the function type can contain numerous type arguments. We separate type arguments with comma (`,`). The stOTTR syntax and the HRS express the function in the previous section as `Function<Function<List<xsd:integer>, Function<xsd:integer, xsd:string>, xsd:string>`.

4.4 Generic type

In Section 4.1, we presented that Frog functions have generic parameters and that function calls have generic arguments. In other words, Frog is a generically typed language. In this section, we show a problem that occurs if Frog were not a generically typed language. Furthermore, we introduce and discuss two solutions to solve this problem, where one of these is now part of the design of Frog. Note that, to illustrate the problem and possible solutions, the discussions in this section assume that Frog is not a generically typed language. Finally, we elaborate on Frog's generic type and define its subtype relations.

Frog is a statically typed language and uses the OTTR type system. However, Frog and OTTR being statically typed introduce a problem. As mentioned in Section 4.1.1, a function call's type equals its affiliated function's return type. OTTR and Frog need to know the type of a function call to perform validation over arguments in instances and function calls separately. Imagine that we have the base function `fn:plus` with type `Function<owl:real, owl:real, owl:real>`, as `owl:real` is the supertype of several numeric types, see Figure 4.11. Furthermore, we want to create a function call on function `fn:plus`, `(fn:plus 1 2)` and use it as an argument in an instance that expects the argument's type to be a subtype of `xsd:integer`. We may assume that this would be valid since the evaluated value of the function call is `3`, which has the type `xsd:integer`. However, the instance considers the function call to have type `owl:real`, which is not a subtype of `xsd:integer`, thus, producing an error message. Consequently, we need to produce a new base function `fn:plusInteger` with type `Function<xsd:integer, xsd:integer, xsd:integer>`, with the same semantics as `fn:plus`.

To further generalise, since the OTTR type system has 18 different numeric types, we would need to produce 18 functions with the semantics of `fn:plus` but with different types to get the desired return type. Appending the numeric functions times and minus, would require Frog to provide 54 functions. Creating these 54 functions is time-consuming and

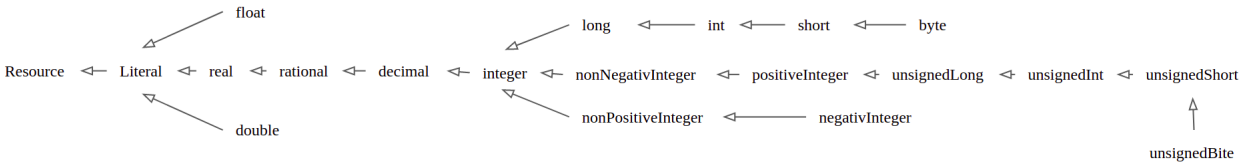

Figure 4.11: Shows the numeric types in the OTTR type system. The figure is an excerpt of the figure of the OTTR type system found in the rOTTR spec[25].

```
fn:plus[ottr:IRI ?name, ottr:IRI ?type]
:: {
    o-rdf:Type(?name, :BaseFunction),
    ottr:Triple(?name, :type, _:type),
    ottr:Triple(_:type, :returnType, ?type),
    ottr:Triple(_:type, :parameterType,
                      (?type, ?type)),
    ottr:Triple(?name, :rule, op:numeric-add),
    ottr:Triple(_:typeVar, :var, ?type),
    ottr:Triple(_:typeVar, :subtypeOf,
                          owl:real),
    ottr:Triple(?name, :typeVars,
                      (_:typeVar))
} .
```

```
# result of expandinc the instance:
# fn:plus(fn:plusInteger, xsd:integer).

fn:plusInteger a :BaseFunction ;
  :rule op:numeric-add ;
  :type [:parameterType (xsd:integer
                         xsd:integer);
         :returnType xsd:integer];
  :typeVars ([:subtypeOf owl:real;
              :var xsd:integer]).
```

Figure 4.12: On the left: an example of how a template generating an `fn:plus` *base function could have looked. One the right: the result of expanding the instance* `fn:plus(fn:plusInteger, xsd:integer)`*, resulting in a Frog base function in the RDF serilaisation.*

contains many repetitions. Hence, not compatible with OTTR's benefit of Don't repeat yourself.

A solution to this problem could be to use the already existing OTTR type, the LUB type. A `LUB<P>` is a subtype of `P`, thus a subtype of every supertypes of `P`. In addition, `LUB<P>` is compatible with every subtype of `P` [21]. We could define the type of the `fn:plus` function to have the type `Function<owl:real,owl:real,LUB<owl:real>>` . Then a function call on `fn:plus` would be compatible with every subtype of `owl:real` and every supertypes of `owl:real`, including `xsd:integer`. However, the function call `(fn:plus 2.3 2.3)` evaluates to the term `4.6` with the type `xsd:decimal`, which is not compatible with `xsd:integer`. Consequently, the LUB type is not a suitable solution to solve the aformentioned problem as it is too general and does not provide any form of restrictions[10].

The previous paragraphs illustrate the need for a restrictive and straightforward approach to create base and Frog functions with the same semantics but different typing. The following paragraphs introduce two different solutions: generating functions through OTTR templates and Frog being a generically typed language.

The first proposed solution is to generate functions through templates that, when expanded, create functions in Frog's RDF serialisation. We suggest a form for import in the Frog and OTTR documents to expand the instances, such as `@importFunction fn:plus(fn:plusInteger, xsd:integer)` in the HRS. Figure 4.12 illustrates the `fn:plus` template and shows the result of expanding the instance above. When importing a func-

[10]Using the LUB type would also not have worked when validating function calls in functions because Frog uses a subtype of validation and not compatible validation. Section 4.5 discusses Frog's validations.

```
def ex:plus2<<?T subtypeOf owl:real>>(?T ?number) -> ?T :: (
    fn:plus<<?T>> ?number (fn:plus<<?T>> ?number ?number)
).
```

Figure 4.13: An example of a generic function, which takes in a number and adds it with itself three times. As the figure shows, we can utilise the generic arguments as parameter types, return type and generic arguments in the function body.

tion in an OTTR or Frog document, we could create function calls on the imported function in the template or function body. The following discussion refers to this solution as *generated functions*.

Another approach is to make Frog a generically typed language, such as Java. As Figure 4.13 illustrates, a function can specify its generic parameters with a variable and be restricted by a subtype-relation on a type. The function call specifies generic arguments that need to be a subtype of the established subtype defined in the corresponding generic parameter. OTTR and Frog would then consider the type of the function `(fn:plus2<<xsd:integer>> 1)` to be `xsd:integer`; beacause substituting the generic variables with these generic arguments into the function `fn:plus2` results to the type `Function<xsd:integer,xsd:integer>`. We formally define this point in Definition 4.4.1.

Definition 4.4.1. *Let* $\boldsymbol{G}$ *be a function with generic parameters* $\boldsymbol{?K_1,\ldots,?K_n}$ *and type* $\boldsymbol{Function<?K_1,\ \ldots\ ,?K_n>}$. *Let* $\boldsymbol{FC}$ *be a function call of function* $\boldsymbol{G}$ *with generic arguments* $\boldsymbol{T_1,\ldots,T_n}$ *in the function or template body of function or template* $\boldsymbol{F}$. *Then* $\boldsymbol{F}$ *regards the type of function* $\boldsymbol{G}$ *to be* $\boldsymbol{Function<T_1,\ \ldots\ ,T_n>}$ *in the context of* $\boldsymbol{FC}$.

We have chosen to make Frog a generically typed language instead of using generated functions. The reason behind this choice is that using the templates would have required the Frog function programmers to create templates based on the RDF syntaxes. However, we assume the Frog function programmer's preferred choice of syntax to be HRS. Another benefit of offering generic functions is that Frog is not dependent on OTTR. On the other hand, the generated functions are dependent on OTTR, as Frog and OTTR produce the generated functions from templates. These templates can contain function calls, creating an unwanted cyclic dependency relationship between OTTR and Frog; because Frog depends on OTTR templates, and OTTR templates depend on Frog function calls.

Moreover, a generic variable P with a subtype relation to the type T is a subtype of T hence also a subtype of T's supertypes. If we introduce another generic variable K, also with a subtype relation to the type T, then we know that P and K have a common supertype. However, that does not mean that these types have a subtype relationship. For instance, if T is `rdfs:Literal`, then it is valid that P is `xsd:string` while K is `xsd:integer`, neither of these types is a subtype of the other type. Consequently, a generic parameter is never a subtype of another generic parameter.

In summary, we have created Frog to be a generically typed language to remove tedious repetitions when creating functions. Using generated functions was also discussed, but we

considered it suboptimal compared to the generic solution. The last paragraph introduced the subtype relation on a generic parameter and the following definitions define the valid use of generic arguments.

Definition 4.4.2. *Let* $\boldsymbol{F}$ *be a function with the generic parameters* $\boldsymbol{?K_1,\ldots,?K_n}$ *with subtypes* $\boldsymbol{P_1,\ldots,P_n}$.
Let $\boldsymbol{FC}$ *be a function call on function* $\boldsymbol{F}$ *with generic arguments* $\boldsymbol{T_1,\ldots,T_n}$.

Then, if $\boldsymbol{FC}$ *is in the function body of function* $\boldsymbol{G}$, $\boldsymbol{FC}$ *is a* ***correct generic typed function call*** *if the following properties holds for every* $\boldsymbol{T_i}$ $1 \leq i \leq n$*:*

- $\boldsymbol{T_i}$ *is a* ***subtype*** *of* $\boldsymbol{P_i}$*, if* $\boldsymbol{T_i} \in$ *OTTR types, or*
- $\boldsymbol{T_i}$ *is a generic parameter defined in the function head of* $\boldsymbol{G}$ *that is defined to be a subtype of* $\boldsymbol{Q}$ *such that* $\boldsymbol{Q}$ *is a subtype of* $\boldsymbol{P_i}$.

Then, if $\boldsymbol{FC}$ *is in a template body,* $\boldsymbol{FC}$ *is a* ***correct generic typed function call*** *if for every* $1 \leq i \leq n$ $\boldsymbol{T_i}$ *need to be a* ***subtype*** *of* $\boldsymbol{P_i}$ *and* $\boldsymbol{T_i} \in$ *OTTR types.*

4.5 Validation

In this section, we discuss the validation, both regarding OTTR and Frog. We discuss how the two terms introduced by Frog require validation when used both in OTTR templates and Frog functions. Moreover, we argue why OTTR require that Frog validates its functions and presents the required validations.

4.5.1 Validation on function call and Function term

Frog has, as mentioned, introduced two new terms to the OTTR type system, namely, the function call and function term. The newly introduced terms differ from the other terms because they have a definition of correctness; see Definitions 4.1.4 and 4.1.5. We have made these definitions to ensure that expansion with function calls and function terms does not produce unwanted errors. For instance, an error would occur if we have a function term referring to a non-defined function. Consequently, OTTR and Frog need to validate the correctness of terms used as arguments in instances and function calls according to the following definitions.

Definition 4.5.1. *An instance or function call* ***has term correctness*** *if every term, with a definition of correctness, used as an argument value in the instance or function call is correct.*

Definition 4.5.2. *A template* ***has term correctness*** *if every instance in the template body and every default value in the template head has term correctness.*

Definition 4.5.3. *A function* ***has term correctness*** *if every function call in the function body has term correctness.*

Checking if an object satisfy any of the definitions mentioned above require OTTR and Frog to perform the following validation on function terms:

- The IRI refers to either a base function or a function in the function library.
- The arity of generic arguments is equivalent to the arity of generic parameters in the affiliated function.
- A generic argument is a subtype of the corresponding generic parameter in the affiliated function.

Additionally, OTTR and Frog are by definition, required to execute the subsequent validations on function call terms:

- The function call name either refers to a defined parameter[11] of the function type or an IRI referring to a base function or a function in the function library.
- The arity of generic arguments is equivalent to the arity of generic parameters in the affiliated function.
- A generic argument is a subtype of the corresponding generic parameter in the affiliated function.
- The arity of arguments is equivalent to the arity of parameters in the affiliated function.
- An argument must be a subtype of the corresponding parameter's type in the affiliated function.

Moreover, the mOTTR specification defines what a semi-valid template library is by listing a set of criteria [21]. We need to add the criteria that every template in a semi-valid template library must have term correctness. Additionally, we need to redefine a valid template dataset, also defined by the mOTTR specification [21], as the following:

Definition 4.5.4. *A **valid template dataset** is a template dataset where:*

- *its template library is valid, and*
- *its set of instances is consistently typed, **has term correctness**, and has referential integrity with respect to the template library.*

4.5.2 Validation on Frog functions

OTTR requires an argument value's inferred type to be compatible with the corresponding parameter's type [21]. As previously mentioned, a function call's type is the type of the affiliated function's return type. Hence, when validating, OTTR uses the return type of a function call's affiliated function. However, a function head's defined return type may not correspond with the actual return value. Function `ex:wrongReturnType` in Figure 4.14 is an example of a function where the return type is incorrect. The function body returns a term with type `xsd:string` while the function head states that the return type

[11] If the function call is in a template, the parameter needs to be defined in the template head. However, suppose the function call is defined in a function body. In that case, the function head need to define the parameter.

```
def ex:wrongReturnType(xsd:String ?str) -> xsd:integer :: (
    (fn:concat ?str ?str)
).
```

Figure 4.14: An example of a function where the type of the value returned by the function and the return type stated in the function head is not compatible (`fn:concat` *is a base funtion taking in two strings and returns a concatinated string).*

is of type `xsd:integer`. Consequently, Frog must validate that the type of the function body[12] is a subtype of the function's return type. Additionally, Frog needs to ensure that a function has term correctness.

Definition 4.4.2 states that if a function call occurs in a function body and contains a generic argument that is a variable, then the generic parameter must be defined by the function. Therefore, Frog is required to validate that every generic argument that is a variable is defined as a generic parameter in the function head. Lastly, Frog must validate that a function head defines every parameter used in the function body. Definition 4.5.5 formally defines a valid function based on the discussions above.

Definition 4.5.5. *A* ***valid function*** *is a function that:*

- *has term correctness, from Definition 4.5.3.*
- *has defined every generic parameter variable used as a generic argument in the function body, from Definition 4.4.2.*
- *has defined every parameter variable used as an argument in the function body.*
- *the type of the function body is a subtype of the function head's return type*

A function call does not only require the affiliated function to be valid but also that every function this function depends on is valid; hence Definition 4.5.6. However, it is not enough to validate that every function that a template library depends on is a dependency-valid function, since functions may be taken in as parameters from ground instances. Consequently, it is impossible to determine which functions to validate from the templates in a template library. Therefore, we suggest that Frog validates that every function in the function library is valid, see Definition 4.5.7.

Definition 4.5.6. *A* ***dependency-valid function*** *is a function where:*

- *it is a valid function, and*
- *every function it depends on is a valid function.*

Definition 4.5.7. *A* ***valid function library*** *is a function library where every function is valid.*

[12]The type of the function body is the type of the function body's function call.

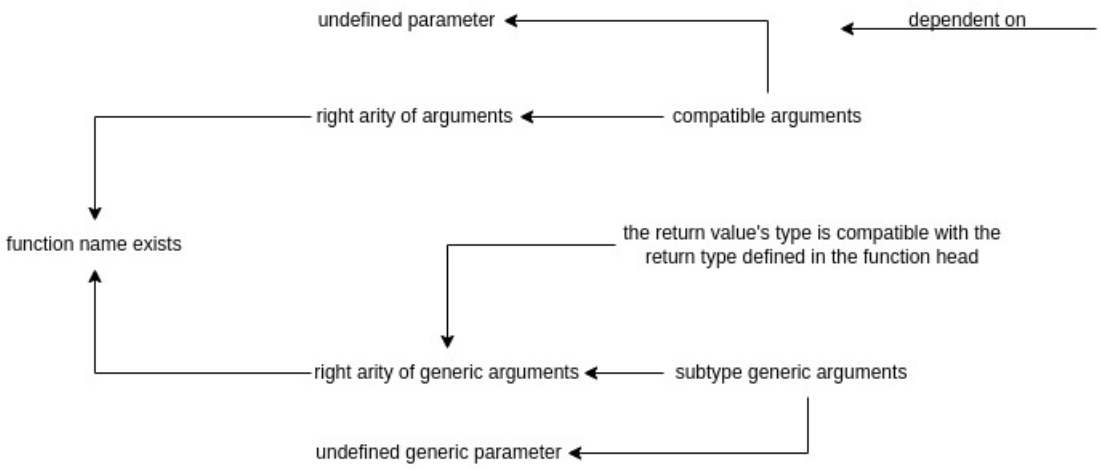

Figure 4.15: The dependencies between the different validations.

4.5.3 The three phases of validating Frog functions

The previous section has discussed what to validate in a Frog function. However, the section does not discuss in which order Frog should perform the validations. The purpose is for Frog to find as many validation errors as possible. However, some validations depends on one or more of the other validations. For instance, to validate that the arity of arguments is correct, Frog needs to know that the affiliated function exists; since Frog compares the arity of arguments to the arity of parameters in the associated function. The same validation must be performed before validating the arity of generic arguments. Figure 4.15 defines the dependency relationship between the validations[13].

Consequently, we have divided the validation into three phases. Phase one validates that the function calls and function terms refer to an existing function. Additionally, phase one validates that there are non-undefined parameters or generic parameters in the function body. The second phase validates that the function calls in the function body have the correct arity of arguments and generic arguments. The third phase validates that the arguments and generic arguments are subtypes of their corresponding parameter or generic parameter and that the return type is valid. The flowchart in Figure 4.16 illustrates the three phases.

4.5.4 Validation warnings

Messages in OTTR have four different severity degrees: info, warning, error, and fatal. By default, OTTR does not expand the instances if a message with the severity degree error or higher is present[14]. The validation previously introduced in this section is necessary for the expansion to proceed, consequently having the severity degree error. However, we wish to provide messages on unused parameters and generic parameters. Having unused parameters and generic parameters requires the function call to provide arguments that the function never uses. Unused parameters and generic parameters do not hinder Frog from being able to evaluate the function calls, hence not hindering the expansion of the

[13]Note that the dependency relationship between validations points is transitive.
[14]In Lutra, we can specify this degree in the execution

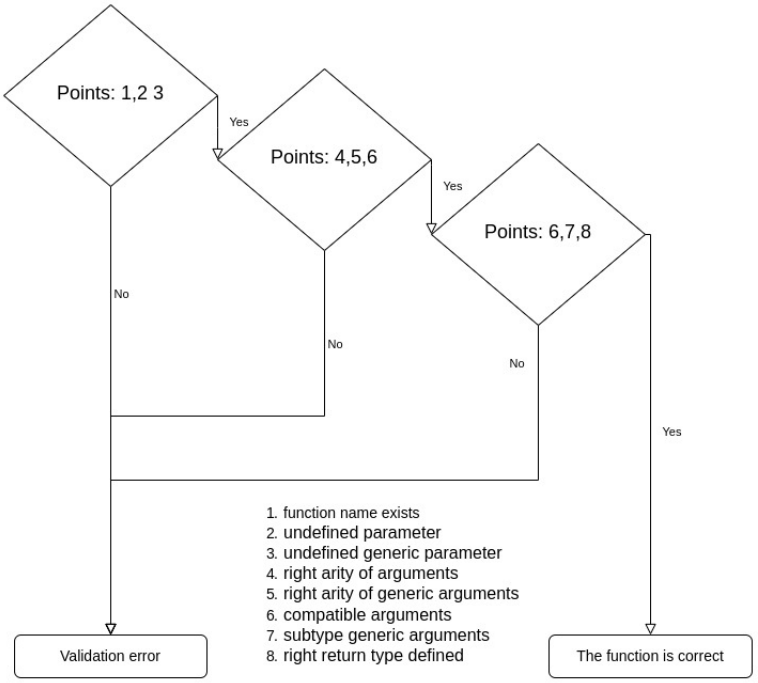

Figure 4.16: The phases and flow of validating a function.

instances. Therefore, Frog provides two validations, one for unused parameters and one for unused generic parameters, that produce messages with the severity degree warning.

4.6 Evaluation

This section discusses which evaluation strategy or approach we believe is most suitable for Frog and OTTR. Firstly, we argue why lazy evaluation is appropriate for Frog, and lastly, we argue that an eager approach is most suitable when OTTR determines when to evaluate a function call.

4.6.1 Arguments for lazy evaluation

In Section 2.3, we introduced eager and lazy evaluations and compared them. When it comes to Frog, the primary purpose of an evaluation strategy is to avoid unnecessary evaluations. Especially regarding base functions, as we assume that computing the base functions is the most time-consuming process when evaluating a function call. Therefore we propose that lazy evaluation is the most suitable evaluation strategy, as opposed to an eager evaluation strategy.

A benefit of using lazy evaluation is that Frog only needs to evaluate a function call when Frog, by definition, needs the evaluated value. Hence, we need to define when we require Frog the evaluate a function call. In Section 4.1.1, we introduced the two types of functions: base functions and Frog functions. As mentioned, base functions performs a single task, such as adding two numbers together. On the other hand, Frog functions are a means of combining base and Frog functions. Since a Frog function only combines other functions, they do not need Frog to evaluate their arguments. However, the base functions

```
def ex:biggestNumberList
    (xsd:integer ?n1, xsd:integer ?n2, List<xsd:integers> ?lst)
                                    -> List<xsd:integers> :: (
    fn:if << List<xsd:integers> >> (fn:greaterThan ?n1 ?n2)
        ?lst
        ()
).
```

Figure 4.17: A function taking in two number and a list, returning the list if the first number is bigger than the second, if not the function returns an empty list.

always require Frog to evaluate at least one of their arguments. The `fn:plus` function, for instance, requires Frog to evaluate both of the arguments to be able to summarise them. On the other hand, the if-function only requires Frog to evaluate the first argument, the boolean, and then evaluate either the second or third argument based on the result of the first argument. Consequently, the base function has a different definition of when to evaluate an argument. Therefore, for Frog to implement lazy evaluation, we chose that Frog only evaluates a function call when a base function requires it.

Moreover, we believe that an eager evaluation approach would have resulted in unnecessary evaluations and calculations on base functions when evaluating function calls. We mostly believe that evaluations over the base functions are time-consuming, particularly the base functions based on XPath; since an implementation of Frog most likely rely on an already existing library to execute the XPath function. Figure 4.18 illustrates the contrast in the number of evaluations performed by Frog to evaluate the function call `(ex:biggestNumberList 5 6 (ex:multiplyNumbers (1)))`[15] in the two evaluation approaches. The reason for the significant contrasts in eager and lazy evaluation, in this case, is that eager evaluation evaluates the arguments before substituting them into the function. The result of eager evaluation evaluating every argument is that the function call `(ex:multiplyNumbers (1))` is evaluated even though it is never used. In contrast, lazy evaluation postpones evaluating the arguments until they are needed, resulting in function call `(ex:multiplyNumbers (1))` never being evaluated. Note that in this case, the amount of evaluation needed when using eager evaluation increases linearly[16] when appending elements to the list argument. In contrast, lazy evaluation has a constant amount of evaluations[17].

An additional benefit of Lazy evaluation is memoisation, which is achievable due to Frog being a purely functional programming language. Memoisation ensures that Frog only evaluates an expression once, retrieving the result from a look-up table if Frog previously has evaluated the function call. The function call `(fn:plus<<xsd:integer>> (fn:plus <<xsd:integer>> 2 2)(fn:plus<<xsd:integer>> 2 2))` is a simple example of why memoisation in Frog is beneficial. Without memoisation, Frog would have needed to

[15] `ex:biggestNumberList` is found in Figure 4.17 while `ex:multiplyNumbers` is found in Figure 4.21.
[16] To be precise 4n + 3, where n is the number of elements in the list.
[17] To be precise 1.

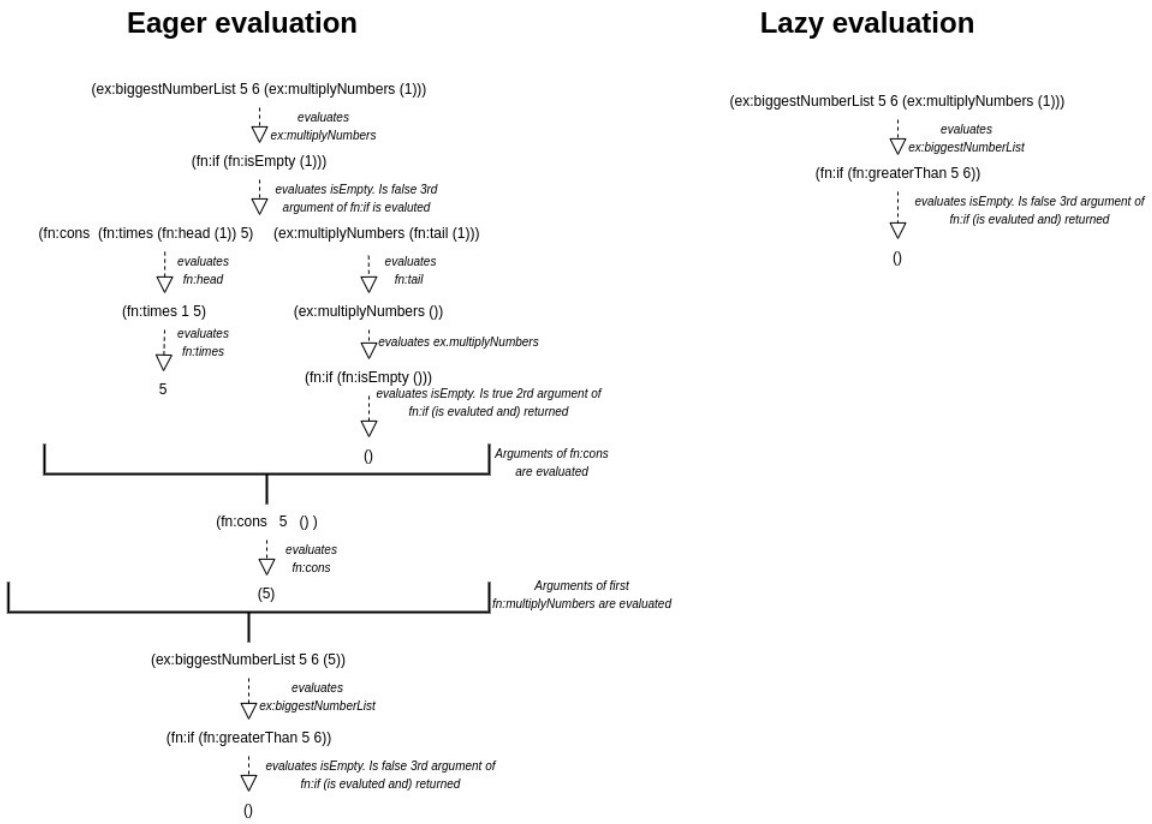

Figure 4.18: Illustrates how the evaluation looks when using eager evaluation compared to lazy evaluation on the function call (ex:biggestNumberList 5 6 (ex:multiplyNumbers (1))), utilising the functions from Figure 4.17 and Figure 4.21. The generic syntax is removed to save place.

evaluate `(fn:plus<<xsd:integer>> 2 2)` twice.

To summarise, we propose that lazy evaluation is the most appropriate evaluation strategy for Frog since lazy evaluation only evaluates the function calls when required and only once due to memoisation.

4.6.2 Evaluation in OTTR

Evaluation in OTTR refers to when OTTR should evaluate the function calls in the instances. We suggest two approaches, firstly, an more eager approach, which evaluates the function calls as soon as OTTR reaches the instance containing the function call and before expanding the instance. In other words, substituting the function call with the evaluated result as soon as possible. Secondly, to evaluate the function calls when they reach a base template, a non-strict evaluation strategy.

When producing OTTR templates, OTTR does not require us to utilise every parameter in the template body, even though OTTR throws a warning message when OTTR finds an unused parameter. Therefore, we can not guarantee that every argument is required for expansion when creating an instance. Figure 4.19 is an example of a template that does

```
ex:person [ottr:IRI ?person, xsd:string ?name,
          xsd:integer ?age, xsd:string ?address] :: {
    ottr:Triple(?person, ex:hasAge, ?age),
    ottr:Triple(?person, ex:hasAge2, ?age),
    ottr:Triple(?person, ex:lives, ?address)
} .
```

Figure 4.19: An example of a OTTR template that contains an unused parameter, namely ?name.

```
ex:person(ex:Per, (ex:concatName "Peter" "Jensen"), 25, "Oslo street 12").
```

Figure 4.20: An example instance of the template found in Figure 4.19, where `ex:concatName` *is the function the first and last name together with space between the names.*

not use every parameter. If we were to utilise the eager approach, OTTR might evaluate unnecessary functions calls, as the template does not require the corresponding parameter to be utilised in the template body. When using an eager strategy, expanding the instance from Figure 4.20 would result in OTTR evaluating the `ex:concatName` function call even though this evaluation is not necessary for the expansion of an instance. On the other hand, using a non-strict evaluation strategy would result in OTTR solely evaluating the function calls when it reaches a base template, hence only evaluating the function call when OTTR needs the evaluated value for an expansion. Expanding the instance from Figure 4.20 would result in OTTR never evaluating the `ex:concatName` function call when utilising a non-strict evaluation strategy.

Nonetheless, using a non-strick approach to evaluate the function call may lead to OTTR executing the same function several times. For example, the instance `ex:person(ex:Peter "Peter Jensen"(fn:minus 25 1))` from the template found in Figure 4.19 would result in OTTR evaluating the `fn:minus` function call twice since the template uses the parameter in two base templates. The number of base templates a function call reaches may be numerous in other cases. On the other hand, the eager strategy would only evaluate the function call once because OTTR would substitute the function call with the evaluate value. However, utilising a non-strick approach would, in practice, result in OTTR only evaluating the function call once since Frog uses memoisation.

Using a non-strict approach in the manner described above destroys OTTR's semantics for the expansion of instances. As described in Section 3.4.4, OTTR does not expand an instance if an argument is `none` and the corresponding parameter is not optional and has no default value. Frog evaluates a function call to `none` if one of the values needed to perform the calculations is none. For example, Frog evaluates the function call `(fn:plus 5 none)` evaluates to `none`. Additionally, there are also other function calls that evaluates to `none`, such as `(fn:head (none 1 2 3))`. However, OTTR does not consider a function call a none value. Therefore OTTR needs to evaluate the function

```
ex:multiplyNumbers(List<xsd:integer> ?lst) -> List<xsd:integer> :: (
    (fn:if<< List<xsd:integer> >> (fn:isEmpty ?lst)
        ()
        (fn:cons<<xsd:integer>>
            (fn:times<<xsd:integer>> (fn:head<<xsd:integer>> ?lst) 5)
            (ex:multiplyNumbers (fn:tail<<xsd:integer>> ?lst))
        )
    )
).
```

Figure 4.21: A function multiplying every number in the list with 5. The result of evaluating the function call (ex:multiplyNumbers (1 2 3)) whould be (5 10 15).

```
#instance

ex:Person(ex:Per, (ex:concatName "Peter" "Jensen"),
         25, (fn:concat "Oslo street" none))

# Result of expanding without hanlding possible none values

ex:Per ex:hasAge 25;
       ex:hasAge2 25.
```

Figure 4.22: Shows the exapnsion with the incorrect non-strict approach. The expansion should have evaluated to an empty set of triples not a set with two tripels.

call to check if the value is `none`. Figure 4.22 illustrates that OTTR expands too much when using a non-strict approach that does not consider that a function call can evaluate to `none`. In the case of Figure 4.22 , OTTR, by definition, should not expand the instance at all because `(fn:concat "Oslo street"none)` evaluates to `none`.

There are two additional reasons why a non-strict approach in OTTR is not suitable. The first reason is strongly associated with the reason described in the previous paragraph. However, this reason focuses more on the default value. OTTR uses the default value if the provided argument value is `none`. As mentioned, OTTR does not consider a function call as `none`. Hence, OTTR does not replace a function call that evaluates to `none` with the default value if the corresponding has defined a default value. Moreover, as described in Section 3.4.4, OTTR generates new instances based on the type of list expander and the list arguments marked with a list expander. Consequently, OTTR needs to know the value the function cmall evaluates to if it is marked with a list expander; since OTTR needs the evaluated value to be able to generate the new instances.

Due to the aforementioned reasons, we argue that a non-strict approach is not suitable for OTTR to determine when to evaluate a function call. Using an eager approach is

inconvenient because OTTR may evaluate unnecessary function calls, as illustrated by Figure 4.19 and Figure 4.20. However, an eager approach does not ruin OTTR's expansion semantics in contrast to a non-strict approach. Therefore, OTTR uses an eager approach to determine when to evaluate function calls.

4.7 Discussion and conclusions

In this chapter, we discussed the design of Frog. We have designed Frog to integrate with OTTR seamlessly. In summary, we made the following choices to ensure this seamless integration:

- OTTR and Frog have the same term and type system
- The syntax of the terms and types are equal in Frog and OTTR.
- A Frog function call is a valid and integrated term in OTTR.
- Frog is a typed language for OTTR to be able to validate that a function call argument is compatible with its corresponding parameter type.
- Frog performs necessary validation on functions to ensure that OTTR's validations still are valid.
- The evaluation of a Frog function call does not conflict with the OTTR expansion semantics due to OTTR using an eager evaluation strategy to evaluate function calls.
- Introducing generics to minimise the number of functions with the same semantics.

Chapter 5

Implementation

In this chapter, we discuss the implementation of Frog in OTTR's reference implementation OTTR, Lutra[1]. Our implementation of Lutra with Frog is publicly available on GitLab[2].The first section shortly introduces the relevant parts of Lutra required for comprehending the subsequent sections. Most sections discuss Frog's implementation in isolation and regard parsing, validation, and evaluation. However, the last section, Section 5.6, describes the changes performed in Lutra's OTTR implementation to integrate Frog. Lutra is a Java implementation of OTTR; hence, we wrote this Frog implementation in Java.

Figure 5.1 illustrates the main steps of Lutra's Frog implementation and how these steps connect. In short, the implementation of Frog starts by parsing Frog functions, either in the RDF syntax or the HRS, into Java objects. We discuss this parsing in Section 5.3. Secondly, the implementation validates that the Function library is valid, as defined by Definition 4.5.7. As illustrated by Figure 5.1, validating the Frog functions is a three-step process:

1. Transforming the function into a RDF query syntax made for querying over, referred to as the *RDF query syntax*.

[1]Lutra is open source and can be found on the following link `https://gitlab.com/ottr/lutra/lutra`.

[2]This is a fork of Lutra, and known issues regarding the Frog implementation can be found on the issues page. `https://gitlab.com/marlenjarholt/lutra-`

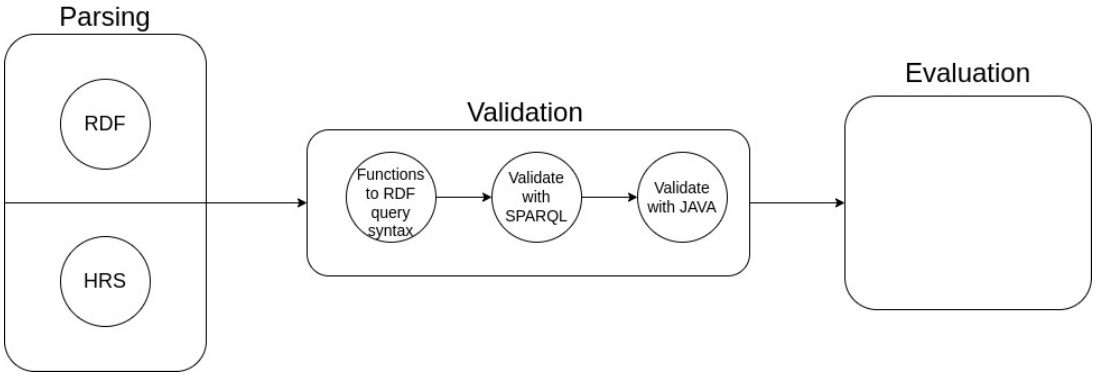

Figure 5.1: The basic flow of Lutra's Frog implementation in isolation.

2. Validating the first and second validation phases with SPARQL.

3. Validating the third validation phase with a pure Java implementation.

Section 5.4 describes the implementation of the validation step. Lastly, section 5.5 introduces the evaluation of function calls.

These three steps depend on their previous step to be able to proceed. Hence, this implies that the validation step requires Lutra to complete the parsing step without causing error messages to proceed. Additionally, Lutra only evaluates function calls if it finishes the validation step without producing any error messages.

The following prefixes will be utilised in this section:

```
@prefix ex: <http://example.xyz/ns/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix fn: <http://ns.frog.ottr.xyz/0.1/function/> .
@prefix : <http://ns.frog.ottr.xyz/0.1#> .
@prefix frog: <http://ns.frog.ottr.xyz/0.1#> .
@prefix ottr:  <http://ns.ottr.xyz/0.4/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix shsh: <http://www.w3.org/ns/shacl-shacl#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
```

5.1 Overview of Lutra's OTTR implementation

As mentioned, we implement our Frog implementation into Lutra, which is the reference implementation of OTTR. This section discusses Lutra's implementation of OTTR templates and instances relevant to the Frog implementation. Lutra offers a command line interface[3] that, among other things, reads templates and instances and expand instances. Similarly to the Frog implementation, the implementation of this command line interface can be considered to consist of three parts: parsing templates and instances into objects, validating the templates, and expanding the ground instances.

Lutra parsers templates and instances in the wOTTR and stOTTR serialisations into objects[4]. This parsing includes parsing terms and types. Since Frog and OTTR have the same term and type system, we can reuse the already established parsing and object implementation on terms and types in the implementation of Frog. Expanding the Lutra implementation with Frog requires parsing and objects on the two new terms, the function term and the function call, the new type, the function type, and lastly, parsers for Frog functions.

Furthermore, the two serialisations of OTTR, wOTTR and stOTTR, apply two different technologies for parsing. On the one hand, wOTTR uses Jena. Jena or Apache Jena is a

[3]The command line interface for Lutra can be found her `https://ottr.xyz/#Lutra`.

[4]Additionally, bOTTR and tabOTTR data from mappings and tabular files into instance objects.

framework built for integrating semantic web and linked data technologies and applications in Java. The Jena framework can parse RDF data in different RDF serialisations and executing SPARQL queries [35]. One of Jena's interfaces allows us to extract specific information from a graph, for instance, extracting every resource of a specific class or every object in a triple with a distinct predicate. wOTTR utilises this interface to extract data on the templates and instances in an RDF file and programmatically verifies that the templates and instances contain the required information in the correct structure[5]. On the other hand, stOTTR uses ANTLR v4[6]. ANTLR v4 is a library made to describe grammar formally and translate grammar into parsing code that is both executable and human-readable [32]. stOTTR, therefore, has its grammar formally described by an ANTLR grammar. The stOTTR specification [20] contains stOTTR's ANTLR grammar. When parsing stOTTR, ANTLR produces error messages when the given file does not match the grammar.

Lutra performs two types of validations: validation on templates and validation on instances during the expansion. The latter is optional and implemented in pure Java. On the other hand, template validation is mandatory, and Lutra implements these validations through queries written for Lutra's self-defined query language. In short, a query is a stream of tuples; where `Tuple` is a class that consists of a map, mapping a string, denoting a variable, to an object. The queries use a `Tuple` object to bind and extract data during querying. To build up a query, we combine different queries through query connectors: and, or, and not. Some frequently used queries are the query that find every template and the query that finds every instance in a template. Code 5.1.1 shows a query that finds every instance in every template by applying the previous queries. Lutra's queries query for errors in the templates and produce error messages, hence validating them.

Code 5.1.1. *An exaple of the query needed to find every instance in every template.*

```
1      /*one stream for each template in the function body
2      binds the string Temp to the template in the tuple*/
3      Query.template("Temp")
4      /*findes every instance in the function body of Temp
5      and bind the instance to Inst (on stream per instance)*/
6          .and(Query.bodyInstance("Temp", "Inst"));
```

Finally, we describe Lutra's implementation for expanding instances, implemented after the definition in mOTTR [21]. Lutra recursively expands the instances until it reaches a base template without a list expander. If an instance contains a list expander, Lutra generates new instances based on the operation and the marked arguments. Example 5.1.1 illustrates the instances generated when applying the `zipMin` operation, which we introduced in Section 3.4. If an instance does not correspond to a base template or contains a list expander, Lutra substitutes the corresponding template's body with this instance arguments, resulting in a set of substituted instances. Code 5.1.2 is a pseudocode for Lutra's expansion of instances.

Code 5.1.2. *A pseudocode for expanding instances in Lutra.*

```
1  FUNCTION expandInstances(instance)
2      template <-gets the template with the same iri as the instance
```

[5] This programmatic verification is based on wOTTR's SHACL shapes and OWL vocabulary.

[6] `https://www.antlr.org`

```
3       IF template do not exist THEN
4           RETURN error
5       IF instance contain none at a non-optional posistion THEN
6           RETURN discard the instance
7       IF instance iri is a base template with no expander
8       OR the instace has ha expander but cannot expand THEN
9           RETURN instace
10      IF instance has list expandet THEN
11          generate instances, one instance per combination of the operator
12          RETURN expandInstaces on all the generated instances
13      ELSE
14          substitute the template's body with the instance's arguments
15          RETURN expandInstances all the substitute instances
```

Example 5.1.1. *An example of the generation of instance produced by the list operation* `zipMin`.

```
zipMin | ottr:Triple(++(ex:name1, ex:name2, ex:name3), ex:relates, ++(1,2)).

#Generate the following two instances
ottr:Triple(ex:name1, ex:relates, 1).
ottr:Triple(ex:name2, ex:relates, 2).
```

5.1.1 Result and MessageHandle

This section briefly introduces the classes `MessageHandler` and `Result`, two central classes in the Lutra implementation for handling messages produced to the user. The implementation of Frog utilises these classes, and they are present in several of the examples in this chapter. The `MessageHandler` and `Result` class rely on the `Message` class. A `Message` object contains a severity, a message and a (possible empty) stack trace. The severity of a message can either be info, warning, error or fatal, where info is the lowest degree of severity, and fatal is the highest. The result of printing a `Message` object is a combination of this message's severity degree and message. Lutra uses `MessageHandler` as a means to collect messages and handle how to print them to the terminal. Moreover, we use the `Result` class when a sequence of operations may result in an error that prodcues messages. The `Result` class works as a wrapper class for the `Optional` class[7]; as the `Result` class consists of an `Optional` value and a trace, which is a possibly empty set of messages produced when working on a `Result` object. The `Result` class implements the `Optional` class's methods, such as `map` and `filter`. In addition to other methods mostly regarding messages and results, such as `mapOrElse`, `aggregate`, and `addMessage`. The `aggregate` method is a static method that takes in a list of `Result` objects and returns a `Result` object of a list (`List<Result<T>> -> Result<List<T>>`); aggregating the `Result` objects into one `Result` object.

5.2 FunctionStore

The `FunctionStore` is a central class in Lutra's Frog implementation. The main objective of this class is to store the functions in the function library and base functions. The `FunctionStore` is central in the parsing and validation step as it offers an interface for, among other things, appending Frog functions and validating the function library. We

[7] `docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Optional.html`

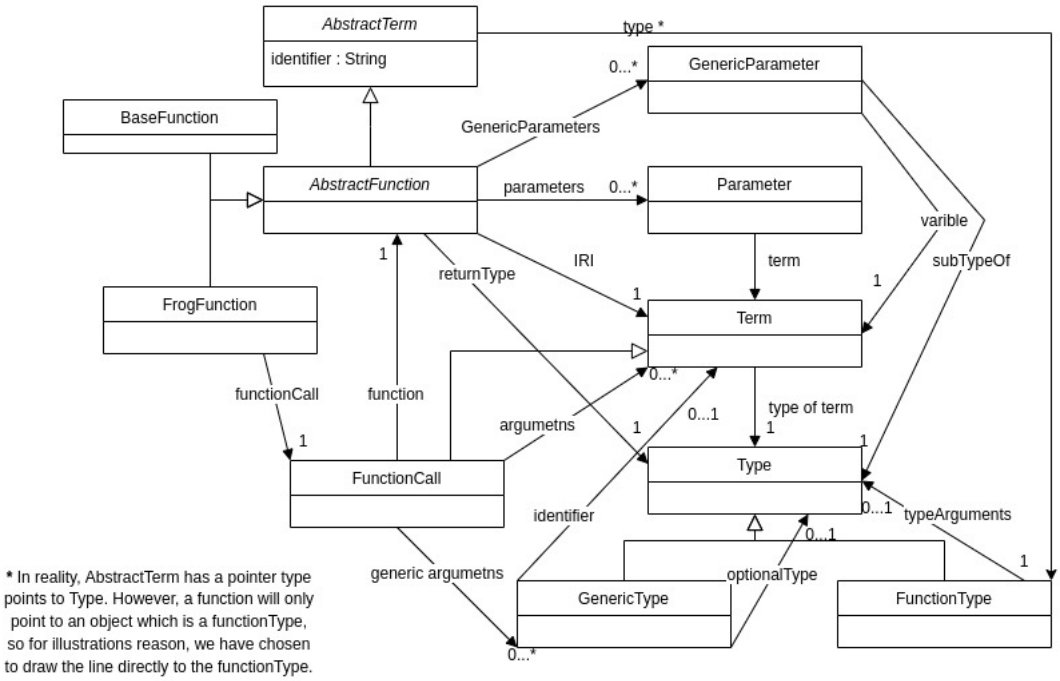

Figure 5.2: A UML diagram of a function. Note that this diagram has removed unnecessary connections and is a simplification of the actual code.

further elaborate on some of the `FunctionStore` methods when describing the different parts of the Frog implementation. For instance, Section 5.4, which contains the method the `FunctionStore` offers for validating a function library.

5.3 Parser

As formerly mentioned, the parsing step in Lutra's Frog implementation consists of transforming functions found in an RDF syntax or HRS document into function objects. Lutra has already created classes for classes for the original terms and types for OTTR[8] before the introduction of Frog. However, we include parsing for the newly introduced terms, the function call and the function term, and the new type, the function type. Figure 5.2 illustrates the classes involved when creating a function object and how they relate to each other.

Furthermore, constructing the objects introduced to Lutra when introducing Frog consists of two phases:

1. Parsing the data from a Frog document, and
2. using a builder[9] to create the objects.

Due to Frog having two serialisations, the first phase has two implementations: one for the RDF syntax and one for the HRS. Section 5.3.1 and Section 5.3.2 discuss the

[8]Term and Type are interfaces in the implementation; however, we have drawn them as classes in the UML diagram

[9]`https://projectlombok.org/features/Builder`

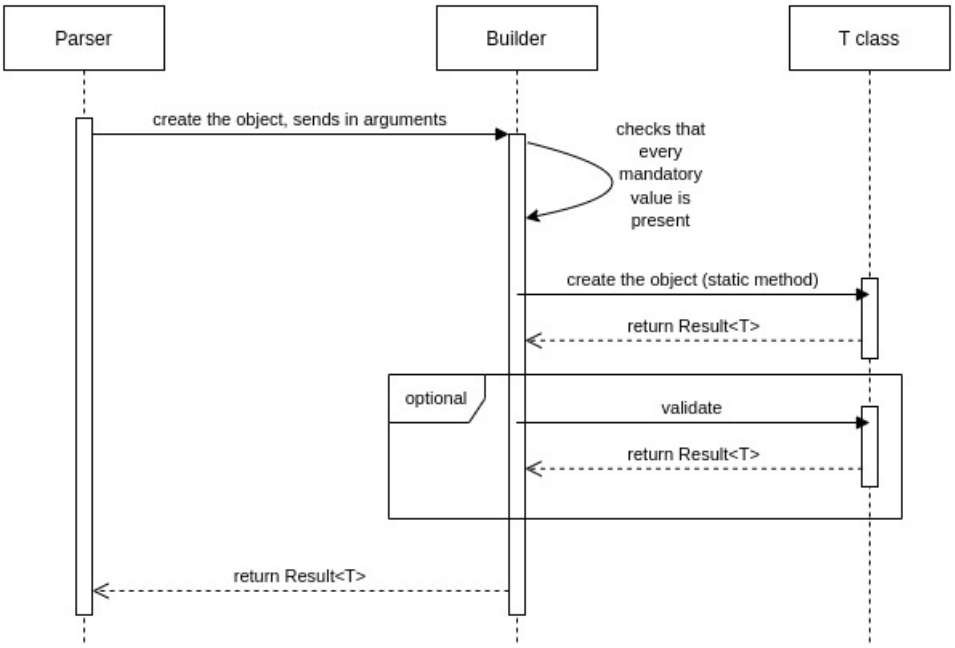

Figure 5.3: A sequent diagram showing the general interaction between a parser, builder and class.

implementation of phase one for RDF syntax and HRS separately. Moreover, the second phase consists of constructing the objects by utilising a builder. There is one builder for each class, and the builders used by the two parsers are equal for both serialisations. Additionally, the builder performs some basic validations, such as ensuring that none of the values are `Null` and that a subtype in a generic parameter does not contain any generic variables. In short, Lutra only performs validations that only require the context of the specific object in the parsing step. We argue that performing validation in phase two of the parsing is more beneficial than phase one because phase two has one implementation compared to phase one with two implementations, resulting in less repetitive code.

The validation performed in the parsing step must not be mistaken with the validation discussed in Section 4.5. In contrast to the validation performed in the parsing step, the validations discussed in Section 4.5 requires knowledge on the context of the function library and not the context of a single object. Section 5.4 describes the implementation of the required validations from the definitions in Section 4.5.

Figure 5.3 illustrates a general interaction between one of the parsers from phase one and the builder from phase two. As we see from the figure, the builder validates that every mandatory argument needed to create the class is present before creating the object. Thereafter, if required, the builder calls the validate method on the object; this step is, however, optional.

5.3.1 RDF Syntax

Similar to wOTTR, Frog's RDF syntax has an OWL ontology describing the vocabulary, and SHACL shapes that define this syntax's grammar. When implementing the RDF syntax's parser, we programmatically produce error messages if the graph does not follow

the constraints set by the OWL ontology and SHACL shapes. Appendix A.1 contains the RDF syntax's OWL ontology and SHACL shapes.

As mentioned, wOTTR utilises Jena to parse an RDF graph containing templates and instances. The wOTTR implementation programmatically validates that the graph follows wOTTR's OWL vocabulary and SHACL shapes during the data extraction. Lutra's wOTTR implementation contains parsers for OTTR's terms and types. Hence, we have chosen to use Jena for parsing Frog functions because we can reuse these parsers. We argue that wOTTR and Frog's RDF syntax using the same parser where it is possible is beneficial since a change in the definitions of terms or types only needs to be implemented in one place instead of two. Thus, the Lutra implementation requires less maintenance.

Lutra's wOTTR implementation contains one parser class for terms named `WTermParser`, and one for types named `WTypeParser`. These parser classes contain one method for each type of term and type, respectively; we refer to these methods as *specific parsing methods*. Additionally, the `WTypeParser` implements an `apply` method that determines which of its specific parsing methods it should use[10]. The `WTermParser`, on the other hand, offers several public static methods to parse a term. These public methods determine which of the term parser's specific parsing method to apply.

The inclusion of Frog requires us to expand the wOTTR's term and type parsers by including the parsing of function terms, function calls and the function type. Hence, we extend the term and type parser with new specific parsing methods: two in the `WTermParser` and one in the `WTypeParser`. Additionally, we modified the `apply` method in the `WTypeParser` and the different public methods in the `WTypeParser` to handle the cases where they should apply the newly introduced specific parsing methods. Figure 5.4 contains the special parsing method included in the `WTypeParser` to parse the function type and the SHACL shape that describes the RDF grammar of a valid function type. The apply method in the `WTypeParser` has confirmed that the first element is `frog:Function` and removed it from the list. Consequently, the `nodes` list only contains the type arguments. Moreover, from the `frog:FunctionTypeShape`, we see that the Function type must have at least one valid type in the Function type list. The special parsing method for the function type ensures this SHACL shape by producing an error message if the `nodes` list is empty. Otherwise, this method parses each element in the `nodes` list and creates a new `FunctionType` object.

Moreover, we implement RDF parsers for the Frog's constructs to parse the Frog function documents in the RDF syntax. We implement these constructs with three new parsers: Firstly, a parser for the generic parameters; secondly, a parser for the parameters; lastly, a Frog function parser for Frog functions. These parsers extract parts of the graph and send these parts to other parsers. For instance, the Frog function parser extracts the return type and uses the type parser to parse the type.

Figure 5.5 shows Lutra's code for parsing the generic parameters and the affiliated SHACL

[10]This `apply` method stems from the `WTypeParser` implementing the `Function` interface:`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html`.

```
ottr:FunctionTypeShape a sh:NodeShape ;
    sh:node shsh:ListShape ;
    sh:property [
        sh:path rdf:first;
        sh:hasValue frog:Function;
        sh:minCount 1
    ],
    [
        sh:path ([sh:oneOrMorePath rdf:rest]
                rdf:first);
        sh:minCount 1;
        #The shape for valid Frog types
        sh:node frog:TypeListShape
    ].
```

```
private Result<Type> parseFunctionType(List<RDFNode> nodes) {
    if (nodes.isEmpty()) {
        return Result.error("Error parsing Function type:
                must have at least one type argument");
    }
    var argTypes = nodes.stream()
            .map(this) //parses the type argument
            .collect(Collectors.toList());
    return Result.aggregate(argTypes)
            .map(FunctionType::new);
}
```

Figure 5.4: On the left: the SHALC shape for the function type, formally defining its RDF grammar. On the right: the special parsing method used to parse a function type.

shapes. The `ModelSelector` class seen in the figure is a class that extracts data from a set of criteria[11]. On lines 5 and 6, the `ModelExtraction` extracts the list object in the triple with the function as the subject and `frog:typeVars` as the predicate. Then this parser parses each of the generic parameters by extracting the generic parameter variables and subtype of relation. A graph that does not contain these triples will result in Lutra producing an error message through the `ModelSelector`, since the method `getRequiredResoucesObject` requires the object and triple to be present. The SHACL shape `frog:GenericParameterShape` defines that a valid generic parameter must contain both values, see lines 12 to 29. Moreover, this SHACL shape also defines that the subtype related node should be a type. Hence, the parser for the generic parameter uses the type parser to parse the type. Additionally, the generic parameter variable should be a blank node. As seen on lines 22 to 24 in the parser code, the parser validates that the object is a blank node and uses the term parser to parse the blank node if the value is a blank node.

5.3.2 Human Readable Syntax

As mentioned, Lutra's stOTTR implementation utilise ANTLR4 to parse a stOTTR document and formally describe stOTTR's grammar[12]. Through this grammar and ANTLR4 generated code, Lutra parses the stOTTR documents. Lutra's stOTTR implementation already supports the parsing of terms and types. Hence, for the same arguments given in the discussion regarding technology for Frog's RDF syntax, we have also chosen to apply the same technology for Frog's HRS as stOTTR, namely ANTLR4.

With ANTLR, we can import another grammar into our grammar. In the case of HRS, it is natural to assume that the HRS grammar imports the stOTTR grammar to be able to reuse the grammar on terms and types. However, when creating a new grammar, ANTLR4 generates two different parsers for the same concepts. In other words, ANTLR4 generates one term parser for Frog's HRS and a different parser for stOTTR. Thus, making it impossible for our Frog implementation to reuse stOTTR's parser. Consequently, we chose to extend the stOTTR grammar with Frog's concepts to be able to reuse the parsing code. Appendix A.2 contains stOTTR's and the HRS's ANTLR4 grammar file,

[11] The ModelSelcector also produces error messages if an expected instance is not present- for instance, that a generic parameter does not contain a relation with the frog:subtypeOf IRI.

[12] `https://dev.spec.ottr.xyz/stOTTR/stOTTR.g4` stores the grammar. Note that this grammar is without the addition of Frog's HRS.

```
frog:TypeVarsShape a sh:NodeShape;
    sh:targetObjectsOf frog:typeVars;
    sh:node shsh:ListShape;
    sh:property [
        sh:path ([sh:zeroOrMorePath rdf:rest]
                 rdf:first);
        sh:node frog:GenericParameterShape;
    ].

frog:GenericParameterShape a sh:NodeShape;
    sh:targetSubjectsOf frog:var, frog:subtypeOf;
    sh:property[
        sh:path frog:var;
        sh:minCount 1;
        sh:maxCount 1;
        #The shape for a generic variable
        #This shape node needs to be a blank node
        sh:node frog:GenericVariableShape;
        sh:name "generic variable";
        sh:message "a generic parameter must
                    be a blank node";
    ],
    [
        sh:path frog:subtypeOf;
        sh:minCount 1;
        sh:maxCount 1;
        #The shape for valid Frog types
        sh:node frog:TypeListShape;
    ];
    sh:name "generic parameter";
    sh:message "a generic parameter needs a
                blank node and type.".
```

```
public class FGenericListParser {

 public static Result<List<Generic>> parseGeneric(Model model,
                                                   Resource function){
    var listOfGenerics = ModelSelector.getOptionalListObject(model,
                                        function, Frog.RDFFrog.typeVars)
        .map(RDFList::asJavaList)
        .mapToStream(ResultStream::innerOf)
        .mapFlatMap(rdfNode -> parseGeneric(model, rdfNode))
        .collect(Collectors.toList());
    return Result.aggregate(listOfGenerics);
 }

 private static Result<Generic> parseGeneric(Model model,
                                             RDFNode generic){
    if (!generic.isResource() && !generic.isAnon()) {
        return Result.error("A generic parameter is defined
                             by a blank node");
    }
    var varValue = ModelSelector.getRequiredResourceObject(model,
                                generic.asResource(), Frog.RDFFrog.var)
        .filterOrMessage(RDFNode::isAnon,
                         Message.error("A generic parameter must
                                        be a blank node"))
        .map(node -> node.asResource().getId().getBlankNodeId())
        .flatMap(WTermParser::toBlankNodeTerm)
        .map(blankNode -> (Term) blankNode);
    var subtypeOf = ModelSelector.getRequiredResourceObject(model,
                          generic.asResource(), Frog.RDFFrog.subtypeOf)
        .flatMap(res -> new WTypeParser().apply(res));
    return GenericBuilder.createGeneric(varValue, subtypeOf);
 }
}
```

Figure 5.5: On the left: the SHALC shape for the generic parameters, formally defining its RDF grammar. On the right: the special parsing method used to parse the generic parameters.

formally defining the grammar of both stOTTR and Frog's HRS.

When reading a Frog function document in the HRS syntax, Lutra, by applying ANLTR4, generates objects in a parse-tree structure to traverse through. ANTLR4 offers two different techniques for traversing parse-trees: parse-tree listeners and parse-tree visitors [32]. Figure 5.6 is an example of a parse-tree produced by ANTLR4 with HRS's grammar. Lutra's stOTTR implements the visitor technique. Consequently, we write Lutra's HRS implementation in the visitor technique as well to be able to reuse code. ANTLR4's visitor technique let us control traversing by explicitly calling methods to visit a context's children [32]. In Figure 5.6, if the code were in the context of the `functionHead`, the context's children would have been: `definition`, `genericParameters`, `parameters`, and `returnType`.

Moreover, ANLTR4 generates a `stOTTRBaseVisitor` class that implements a visit method for each rule in our grammar. An example of a rule in stOTTR's and HRS's grammar is `functionType`. Figure 5.7 shows the function type rule on lines 10 to 12. To decide what should happen when executing these visiting methods, we create classes that extend the `stOTTRBaseVisitor` and override the visit methods. The stOTTR implementation contains parsers that override the visit methods on terms and types, namely the `STermParser` and `STypeParser`. However, similar to the RDF syntax, we extend these parsers by appending overriding visit methods for the function type, function call and function term. Figure 5.7 illustrates the changes and additions in the grammar to include the function type and the overridden visit method Lutra implements to parse function types in the `STypeParser`. The code extracts every type of child of the function type's context, resulting in a list of type contexts, on line 2. We parse each of these type contexts through the `visit` method for types. When Lutra finishes parsing the type contexts, we

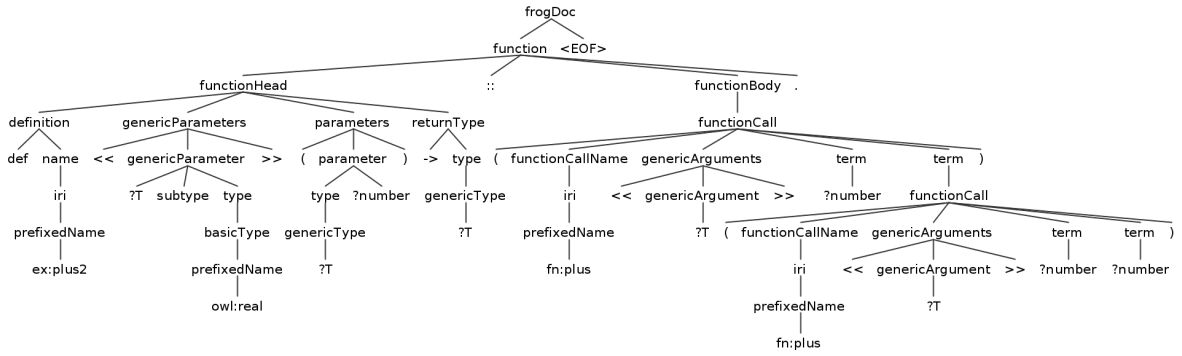

Figure 5.6: The generated parser-tree made by ANTLR4s java implementation on the function in Figure 4.13.

```
type
 : basicType
 | lubType
 | listType
 | neListType
 | functionType
 | genericType
 ;

functionType
 : 'Function<'((type ',')* type) '>'
 ;
```

```
public Result<Type> visitFunctionType(FunctionTypeContext ctx) {
    var types = ctx.type().stream()
            .map(this::visitType)
            .collect(Collectors.toList());
    var aggrRes = Result.aggregate(types);
    return aggrRes.flatMap(aggr -> Result.of(new FunctionType(aggr)));
}
```

Figure 5.7: On the left: the ANTLR4 grammar rule for the function type. On the right: the overridden visit method used to parse a function type.

use them to create a `FunctionType` object.

Similar to the RDF syntax implementation, we implement three parser classes for the Frog constructs to parse: a generic parameter, a parameter, and Frog functions. These parsers extract specific contexts and utilise other parsers to parse the contexts. The Frog function parser, for example, extracts the name of the function and validates that the value is present in the parse-tree before utilising the term parser to parse the name into a `Term` object. Furthermore, Figure 5.8 shows the ANTLR4 grammar for the generic parameter and the parser class, `FGenericParameterParser`, that parses a generic argument. This class implements a visit method to visit a generic parameter context and utilises the term parser to parse the blank node and type parser to parse the subtype relation. Similar to how the generic parameter parser for the RDF syntax appplies its related term and type parsers.

5.4 Validation

Section 4.5 discusses the need for Frog to validate that a function library is a valid function library. From the definition of a valid function library, Definition 4.5.7, we assemble the following list of required validations that Lutra's Frog implementation must validate on every Frog function:

1. Every function call's name in the function body must refer to a defined parameter variable of the function type, a base function, or a function in the function library.

```
    ;

genericParameterList
    : '<' '<'
        ((genericParameter ',')* genericParameter?)
      '>' '>'
    ;

genericParameter
    : Variable 'subtypeOf' type
    ;
```

```
public class FGenericParameterParser extends SBaseParserVisitor<Generic> {

    private final STypeParser typeParser;
    private final STermParser termParser;

    public FGenericParameterParser(STermParser termParser) {
        this.termParser = termParser;
        this.typeParser = new STypeParser(termParser, false);
    }

    public Result<Generic> visitGeneric(GenericParameterContext ctx) {
        return GenericBuilder.builder()
                .term(parseTerm(ctx))
                .type(parseType(ctx))
                .build();
    }

    private Result<Term> parseTerm(GenericParameterContext ctx) {
        var label = termParser.getVariableLabel(ctx.Variable());
        return termParser.toBlankNodeTerm(label)
                    .map(t -> (Term) t);
    }

    private Result<Type> parseType(GenericParameterContext ctx) {
        return ctx.type() != null
                ? typeParser.visit(ctx)
                : null;
    }
}
```

Figure 5.8: On the left: the ANTLR4 grammar rule for the the generic parameters. On the right: the overridden visit method used to parse a generic parameter. The class `SBaseParserVisitor` *extends* `stOTTRBaseVisitor`*, wrapping the generic argument into the Result class (*`class SBaseparserVisitor<T> extends SBaseParserVisitor<Result<T>>`*).*

2. The function must define every variable used as an argument in the function body as a parameter in its function head.
3. The function must define every variable used as a generic argument in the function body as a generic parameter in its function head.
4. The IRI in a function term must refer to a function in the function library or a base function.
5. The arity of parameters in a function call in the function body must equal its affiliated function's arity of parameters.
6. The arity of generic arguments in a function call or function term used as an argument in the function body must be equal to its affiliated function's arity of generic parameters.
7. For every argument in a function call in the function body, its type must be a subtype of its corresponding parameter's type.
8. Every generic argument in a function call or a function term in the function body must be a subtype of its corresponding generic parameter.
9. The return type of the function body is a subtype of the function head's specified return type.

Additionally, as presented in Section 4.5.3, we want to produce warning messages for unused variables. Thus, the Lutra implementation needs to check that for every function in the function library:

10. Every parameter defined in the function head is used in the function body.
11. Every generic parameter defined in the function head is used in the function body.

This section discusses the technologies used, the implementation of the validations, and the execution of these ten validations.

5.4.1 Technology

Lutra has a self-made query language, as introduced in Section 5.1. However, we argue that there are several benefits of applying established technologies rather than an Lutra's query language. Hence, we chose to apply SPARQL queries to perform the validations. Using SPARQL for validating the nested type structure, however, proved difficult. Consequently, we chose to utilise a pure java implementation for the validation regarding types. Section 6.1.1 discusses in more detail why we argue that applying SPARQL is more beneficial than Lutra's query language and a SPARQL limitation resulting in the need for a pure Java validation for validating types. The subsequent two sections describe the validation implementation with SPARQL and Java separately.

5.4.2 SPARQL

This section discusses the part of the validation implementation that applies SPARQL as its technology. Firstly, we introduce a special RDF syntax which makes it easier to query compared to Frog's RDF syntax, namely the *RDF query syntax*. Secondly, we discuss the two types of SPARQL validations and discuss one query per type. Lastly, we examine our implementation for executing the SPARQL validation queries in Lutra.

RDF query syntax

Frog's RDF syntax utilises RDF lists as a means to represent several of Frog's constructs. As Section 4.2 discuss, utilising a list structure is beneficial since it is an established structure that we can write compact in the Turtle serialisation of RDF. However, this structure results in a loss of metadata, making it harder to construct SPARQL queries that are compact and easy to read and write. Consequently, we chose to introduce a new syntax, *the RDF query syntax*, made to be easy to construct queries over that are readable. Section 6.1.2 discuss the benefits and disadvantages of using and introducing a new syntax for the SPARQL queries compared to utilising the existing RDF syntax.

We tried to make the RDF query syntax resemble Frog's RDF syntax by reusing several properties described in Frog's RDF syntax. However, there are two significant differences between the RDF query syntax and Frog's RDF syntax. Firstly, for the reasons mentioned in the previous paragraph, we have replaced the RDF lists used to represent Frog and OTTR concepts, such as function calls, parameters and generic arguments[13]. The RDF query syntax represents these concepts with blank node structures that, most importantly, explicitly state the indexes, resulting in a graph in the RDF query syntax containing more metadata than a graph in Frog's RDF syntax. Secondly, the RDF query syntax

[13]The RDF query syntax still represents a list term with an RDF list.

```
ex:FunctionName frog:parameter
    [frog:index 1;
    frog:parameterType xsd:integer;
    frog:var _:number1],
    [frog:index 0;
    frog:parameterType xsd:integer;
    frog:var _:number2].
```

Figure 5.9: Illustrates how a parameter looks in the RDF query syntax

```
[] frog:of ex:FunctionCallName;
    frog:arg [frog:index 0
              frog:val ex:value0];
    frog:typeArg [frog:index 1
                  frog:type xsd:string],
                  [a frog:GenericType;
                  frog:index 0
                  frog:var _:genricVariable].
```

Figure 5.10: Illustrates how a function call, arguments and generic arguments looks in the RDF query syntax.

more closely resembles the structure of Frog functions shown in Figure 4.3, having a more precise separation of the function head and body. An example of such a change is that the RDF query syntax represents a parameter with a blank node containing both the parameter type and the parameter variable; in contrast to containing them in two separate lists. Figure 5.9 illustrates the syntax of parameters in the RDF query syntax[14].

The RDF query syntax represents a function call in a particular blank node structure. Figure 5.10 illustrates a function call in the RDF query syntax. As shown in this figure, a function call blank node relates to its name through the `frog:of` predicate and the arguments and generic arguments through the predicates `frog:arg` and `frog:typeArg`.

Queries

The SPARQL validation queries aim to find cases in the functions that do not uphold the definition of a valid function. In other words, the queries find violating functions; thus, every result of a query is a violation of the definition of a valid function, Definition 4.5.5. For instance, finding a function that utilises a variable used as a generic argument in the function body that the function head does not define. As mentioned, the validation implementation with SPARQL validates every validation that does not depend on types; hence, validating phases one and two from the validation phases described in Section 4.5.3. Consequently, we have written seven SPARQL queries, including the warning validations.

In short, we categorise the validation queries into two types: *object queries* and *arity queries*. The validation queries validating the correct arity of arguments and generic arguments are arity queries, while the rest of the queries are object queries. The object queries consist of two stages: Firstly, to find the object we are validating on, and secondly, to remove the objects with a correct definition or use. The second step mainly consist of NOT EXIST clauses. On the other hand, the arity validation queries consist of four stages. Firstly, find all function calls and/or function terms in function bodies. Secondly, find the arity of arguments or generic arguments in these function calls. Thirdly, find the

[14]Note that the parameters explicitly state their indexes. To extract the index of a parameter in Frog's RDF syntax would have required a subquery.

arity of parameters or generic parameters in the functions, in other words, the number of expected arguments or generic arguments. Lastly, utilising a FILTER clause to remove the matches where the function call arity of arguments or generic arguments equals the affiliated function's expected arity.

Moreover, there exist two pairs of reverse validation queries: parameter variable and unused parameter variable, and undefined generic variable and unused generic variable. We have named these pairs of queries reverse validation queries because the first query's first step is logically equal to the second query's second step, and the second query's first step is logically equal to the first query's second step; thus, the first and second step in these pairs of queries are revers of each other. Consequently, understanding one query in a pair of reverse validation queries makes it easier to understand the other because they consist of the same logic.

Definition 5.4.1. *For a pair of queries to be **reverse validation queries**, they must fulfil two criteria; we use **X** and **Y** to represent the queries in the pair of reverse object queries:*

1. *The first stage in **X** must contain the same query patterns as **Y**'s second stage. Consequently, **X**'s first stage pattern is equal to the pattern found in Y's NOT EXIST clause. If **X** first stage pattern contains UNION clauses, then each UNION pattern matches the pattern found in a NOT EXIST clause in the second stage of **Y**.*

2. *The second stage in **X** must contain the same query patterns as **Y**'s second stage. The translation between stage one and stage two is equivalent to point one.*

In this section, we present two queries: one object query and one arity query. Appendix B stores all the SPARQL validation queries utilised by Lutra to validate the Frog functions. Firstly, we present Query 5.4.1, the none existing function query. This query finds every function call in a Frog function where the function call's name does not defined in the function library or is a parameter variable of the function type. Moreover, this query is an object query, thus, consisting of two stages. Firstly, the query finds the object it should validate; in this case, every function call in every Frog function. We use property paths to recursively find every function call in the function body, as seen on lines 5-6. This recursive use of property paths ensures that the none existing function query not only finds the outer function call but also the function calls used as arguments inside another function call. Secondly, we need to find the incorrect use of a function call name. From Definition 4.1.4, we have that a function call's name can be a base function, a Frog function, or a parameter variable defined as the function type in the function body. Consequently, we have created two NOT EXIST clauses, as seen on lines 10-22. The first NOT EXIST clause removes any pattern where the function call's name is an IRI and a defined function[15]. Furthermore, the second NOT EXIST clause removes every pattern where the function call's name is a variable[16], and the function head defines the variable as a parameter of the function type. Consequently, this SPARQL query finds every pattern in the graph of a function containing one or more function calls where the function call's name is not correctly defined.

[15]Base or Frog function.

[16]Which the RDF query syntax express with a blank node similar to Frog's RDF syntax.

Query 5.4.1. *The validation query used to extract the function calls that does not use an exisiting function (either from the function library or a base function) or a parameter of the function type.*

```
1  SELECT DISTINCT ?functionName ?functionCallName
2  WHERE {
3      #finds everything used as a function call
4      #name in the function body
5      ?functionName a frog:Function;
6         frog:body/(frog:arg/frog:val)*/frog:of ?functionCallName.
7
8      #removes all mathces where the function call
9      #name is an IRI and the IRI is of type function
10     FILTER NOT EXISTS {
11             ?functionCallName a frog:Function.
12             FILTER isIRI(?functionCallName)
13     }
14     #removes all mathces where a blank node is used as
15     #function is defined as a parameter of type function
16     FILTER NOT EXISTS {
17             ?functionName frog:parameter
18                           [frog:var ?functionCallName;
19                            frog:parameterType [a frog:Function]
20                           ]
21             FILTER isBlank(?functionCallName)
22     }
23 }
```

Finally, we present Query 5.4.2, the incorrect arity of arguments query. As suggested by the name, this query is an arity query that extracts every function call in every function where the arity of arguments are unequal to the arity of parameters in its affiliated function. Moreover, this query consists of four stages because it is an arity query. Firstly, to find the object we are validating. In this case, every function call in every function body. This step is the same as the first step in Query 5.4.1; hence these steps are almost equivalent. However, in the incorrect arity of arguments query, we need the blank node representing the function call in addition to its name. Therefore, as seen on lines 3-5, this query finds the function call and the function call's name in two steps. Secondly, this query needs to find the arity of arguments in the function call. To find this arity, we have created a subquery, as seen on lines 7-16. This subquery extracts every argument of every function call in the graph and uses aggregation to count the argument number. Furthermore, we have used an OPTIONAL clause to match the argument pattern since some function calls may have an empty set of arguments.

Thirdly, in the third step, the query finds the arity of parameters in every function and parameter of the function type. As seen on lines 18-48, we have created two subqueries to find the arity: one that counts the parameters in the function type parameters and one that counts the parameters in functions. We constructed these queries similar to the subquery in the second step. Note that the subquery for the function type parameter uses the `frog:index` predicate since the function type only contains the index triple on

the parameters. Moreover, the result of these two subqueries is unified with the UNION clause. Lastly, in the fourth step on line 49, we use the FILTER clause such that only the pattern matches where the arity of the received number of arguments differs from the number of expected arguments. Note that SPARQL performs a inner-join between the second and third step.

Query 5.4.2. *The validation query to find every parameter variable used in the function body that is not defined by the affiliated function's head.*

```
1  SELECT DISTINCT *
2  WHERE{
3      ?functionName a frog:Function;
4          frog:body/(frog:arg/frog:val)* ?functionCall.
5      ?functionCall frog:of ?functionCallName.
6
7      { #finds how many arguments the function call has
8          SELECT ?functionCall (COUNT(?rec) AS ?received)
9          WHERE {
10             ?functionCall frog:of [].
11             OPTIONAL{
12                 ?functionCall frog:arg ?rec.
13             }
14         }
15         GROUP BY ?functionCall
16     }
17     #finds how many parameters the function has
18     { #if the function is defined with a parameter variable
19         {
20             SELECT ?functionCallName (COUNT(?exp) AS ?expected)
21             WHERE {
22                     [] a frog:Function;
23                         frog:parameter [
24                             frog:var ?functionCallName;
25                             frog:parameterType ?parType
26                         ].
27                     ?parType a frog:Function.
28                     OPTIONAL { #finds the parameters
29                                #of a parameterfunction
30                         ?parType frog:argType/frog:index ?exp
31                     }
32             }
33             GROUP BY ?functionCallName
34         }
35     } UNION { #if the functon is defined with a IRI
36         {
37             SELECT ?functionCallName (COUNT(?exp) AS ?expected)
38             WHERE {
39                 ?functionCallName a frog:Function.
40
41                 OPTIONAL{
42                     ?functionCallName frog:parameter ?exp.
```

```
43                  }
44                  FILTER isIRI(?functionCallName)
45              }
46              GROUP BY ?functionCallName
47          }
48      }
49    FILTER(?received != ?expected)
50 }
51 ORDER BY ?functionName ?functionCallName
```

Lutra's execution

Our implementation for executing and creating error messages in Lutra consists of two parts. Firstly, we create a Jena model[17] storing the base and Frog functions in the RDF query syntax. Secondly, we use Jena's query library to read the queries and execute them. We have constructed one execution class for each query, referred to as a query class. A query class implements the interface `FunctionCheck`. This interface extends Java's interface `Function`[18], taking in a Jena model and returning a `MessageHandler` containing the messages produced by the query. Thus, the FunctionCheck interface consists of an apply method taking in a Model and returning a `MessageHandler`.

Code 5.4.1. *The check class that executes and produces error messages for the none existing function query, query 5.4.1. The parameter resultSet contains the result from executing the query, one row for each match.*

```
1  public class CheckFunctionExists implements FrogCheck {
2   private static final String queryFile = "checkFunctionExists.rq";
3
4   @Override
5   public String getValidationFile() {
6     return queryFile;
7   }
8
9   public MessageHandler errorMessage(ResultSet resultSet) {
10     var msgs = new MessageHandler();
11     var list = ResultSetFormatter.toList(resultSet);
12     list.forEach(querySolution -> {
13       var functionName = querySolution.get("functionName").toString();
14       var functionCallName = querySolution.get("functionCallName");
15
16       var errMessage = functionCallName.asResource().isAnon()
17           ? "The variable " + Frog.getVarNameFromUniqueId(functionCallName.asResource())
18             + " is not a function it's however used as a function in " + functionName
19           : "The function " + functionCallName + " which is used in "
20             + functionName + " does not exist";
21       msgs.add(Message.error(errMessage));
22     });
23    return msgs;
24   }
25 }
```

Additionally, the interface consists of two other methods: `getValidationFile`, returning the query file, and `errorMessage`, producing a `MessageHandler` with the validation's

[17] A Jena model is an RDF graph.

[18] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html`

error messages. The `FunctionCheck` interface's apply method is a default method written in the interface, which reads in the query file, executes the query, and creates the `MessageHandler` by applying the `errorMessage` method on the result of executing the query. In Code 5.4.1, we see an example of a query class, namely the query class for the none existing function query. Thus, this query executes and creates error messages for Query 5.4.1. The query class in Code 5.4.1 produces the following error messages on the function in Figure 5.11:

```
[ERROR] The variable ?number is not a function
        it's however used as a function in ex:minus2
[ERROR] The function fn:minklus which is used in
        ex:minus2 does not exist
```

```
def ex:minus2(xsd:integer ?number) -> xsd:integer :: (
    fn:minklus<xsd:integer> 5 (?number 1 2)
).
```

Figure 5.11: An example of a function that utilises undefined parameters in the function body.

5.4.3 Java

As mentioned, a pure Java implementation validates typing:

- Arguments are subtypes of their corresponding parameter.
- Generic arguments are subtypes of their corresponding generic parameter.
- The function body's type is a subtype of the function head's return type.

These three evaluations equal the evaluation in the third validation phase as described in Section 4.5.3. Moreover, the Java validations follow a similar pattern. More precisely, applying a method in the `Function` interface that takes in the `MessageHandler` and append error messages on this `MessageHandler` when a validation error occurs. Either this `Function` class method creates and appends the messages, or this method calls a method offered by the `FunctionCall` class that creates and appends the messages.

The similarity between the three validations is that they validate that something is a subtype of something else. Lutra's `Type` interface offers a method, `isSubTypeOf`, that takes in another type and returns true if this type is a subtype of the other type, otherwise false. The type classes contains an implementation of the `isSubTypeOf` method after the definitions of the subtype relationship in OTTR, defined by the mOTTR [21] and rOTTR [25] specifications. When including Frog into OTTR, we introduced the new function type. In Lutra's implementation, this addition has resulted in the class `FunctionType`, which implements the `Type` interface. Consequently, we need to implement the `isSubTypeOf` method in this class. After Definition 4.3.2, we implement this `isSubTypeOf` method, as seen in Code 5.4.2[19].

[19]The functionType variable is the list of type arguments.

Code 5.4.2. *The method* `isSubTypeOf` *implemented in the FunctionType class. Implemented from the formal definition of a subtype relatonship.*

```
@Override
public boolean isSubTypeOf(Type other) {
    if (other.equals(TypeRegistry.TOP))return true;
    if (!(other instanceof FunctionType))return false;
    var functionTypeOther = (FunctionType) other;
    if (functionTypeOther.functionType.size() != functionType.size()){
        return false;
    }

    for (int i = 0; i < functionType.size() - 1; i++) {
        var otherType = functionTypeOther.functionType.get(i);
        if (!otherType.isSubTypeOf(functionType.get(i)))return false;
    }
    var last = functionType.size() - 1;
    var otherLastType = functionTypeOther.functionType.get(last)
    return functionType.get(last).isSubTypeOf(otherLastType);
}
```

Moreover, when validating the third validation phase, we assume that the validation in the first and second phase is correct[20]. Thus, the Java validation implementation does not check, for instance, that the number of arguments and the number of the parameters in its affiliated function are equivalent; it assumes that they are. Consequently, making the Java implementation shorter and easier to understand. The rest of this section shortly describes each of the Java validations.

Arguments are subtypes

To validate that every argument in a Frog function's body is a subtype of its corresponding parameter, the `Function` class offers the method `validateArguments`. As depicted in Figure 5.12, the `Function` class uses the `validateArguments` method in the `FunctionCall` class. This method validates that every argument is a subtype of its corresponding parameter by extracting the type of the term and using the `isSubTypeOf` method on it with its corresponding parameter's type as the argument. A call on the `isSubTypeOf` method resulting in the value false will produce an error message. Lastly, the `FunctionCall` class's `validateArguments` method calls this method on all of its arguments of type `FunctionCall`. Hence, Lutra validates the arguments in the function body recursively.

Moreover, to perform the validation correctly in regards to function calls with generic arguments, we substitute the generic parameters with their generic argument, similar to how Definition 4.4.1 defines how the templates and functions interpret the type of a function term with generics. Consequently, when validating the function call `(fn:plus<<xsd:integer>> 5 5)`, where the function `fn:plus` has the type `Function<?T, ?T, ?T>`, we substitues the `?T` with `xsd:integer`; resulting in the following list containg the parameter types `(xsd:integer, xsd:integer)`. The `validateArguments` uses this substituted list when validating the arguments.

[20]These phases validate that every expected value is present and exists.

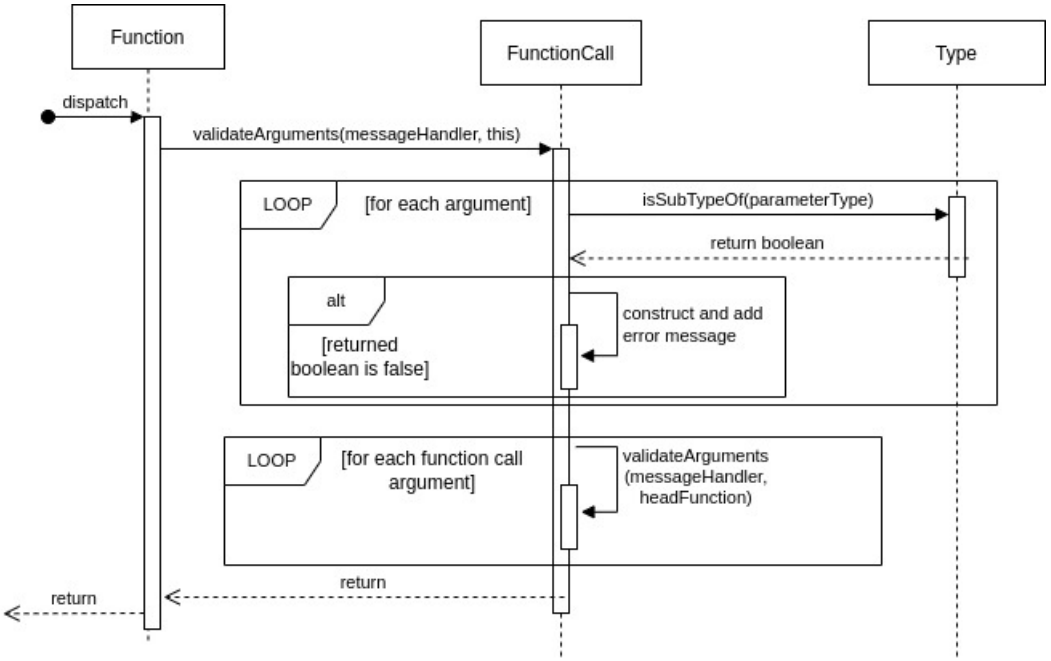

Figure 5.12: A sequent diagram representing the validation of arguments in a Frog function body.

Generic arguments are subtypes

The `Function` class offers the method `validateGenericArgument`, which validates that every generic argument in the function's body is a subtype of its corresponding generic parameter. The validation of the generic arguments is similar to the validation of arguments; since the Function class's method, `validateGenericArgument`, calls on the `FunctionCall` class's method `validateGenerics` that recursively validates every generic argument in the function body, as seen in Figure 5.13. The `validateGenerics` method is a method implemented in the `Term` interface. This method is by default empty; however, implemented in the `FunctionTerm`, `FunctionCall`, and `ListTerm` class since a `ListTerm` can contain function calls and function terms. As seen in Figure 5.13, the `validateGeneric` method in the `FunctionCall` class consist of two steps: validating its generic arguments and validating every argument's generic arguments by calling on the `validateGeneric` method. Hence this method validates not only the function calls but also the function terms.

Moreover, the implementation of the `validateGeneric`, in both the `FunctionTerm` and `FunctionClass` class, utilises the `Function` class's method `validateGenerics`, which takes in a list of generic arguments and validates that the arguments are a subtype of the function's parameters. The `genericMap` argument used in Figure 5.13, is a map for a generic variable to its type.

Compatible return type

Finally, we discuss the implementation of the return type validation. The `Function` class offers this validation through the `validateReturnType` method. This method uses the

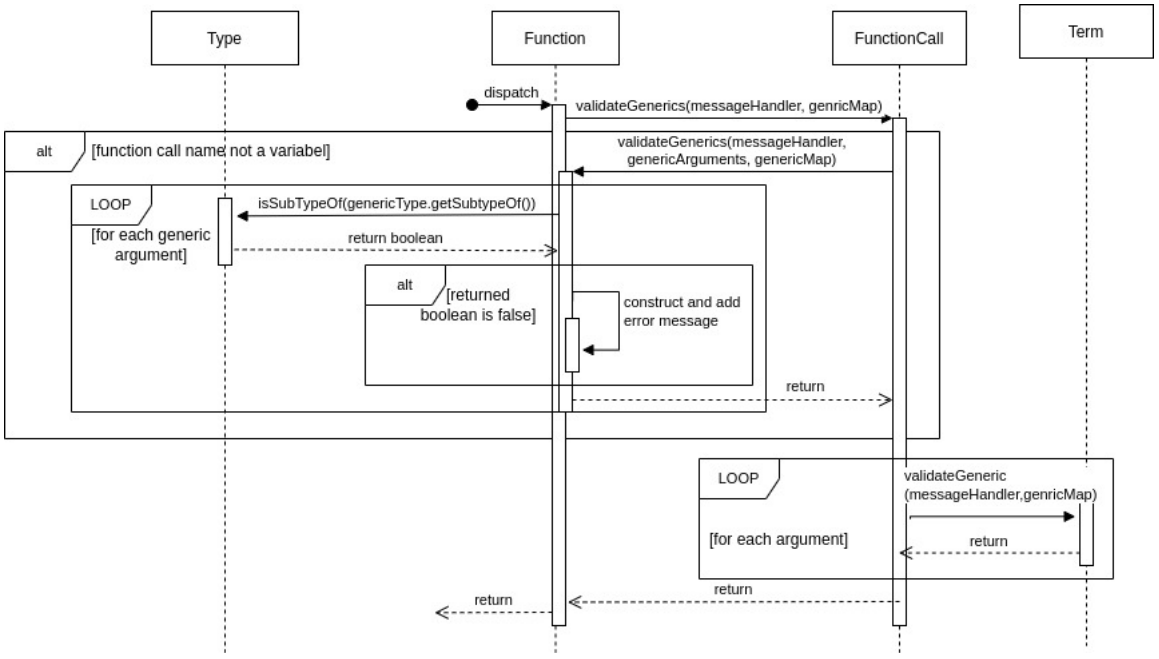

Figure 5.13: A sequent diagram representing the validation of the use of generic arguments.

`isSubTypeOf` method on the function body's type[21] and sends in the function's return type as the argument. The method produces an error message if the `isSubType` call returns the value false, as depicted in Figure 5.14.

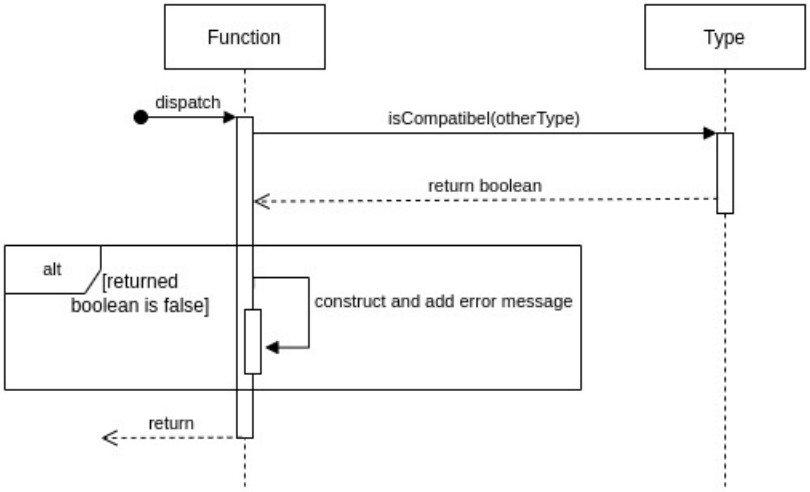

Figure 5.14: A sequent diagram representing the validation of the use of return type.

5.4.4 Execution of the validation

To execute the validation, the `FunctionStore` offers a method `validateFunctions`, seen in Code 5.4.3, which validates the functions in the `FunctionStore` and returns

[21] The function body is a function call; thus, the function body's type is the outermost function call's type.

Code 5.4.3. *Shows the method* `validateFunctions` *in the* `FunctionStore` *class, which validate the the functions are correct.*

```
1  public MessageHandler validateFunctions(PrefixMapping prefixMapping) {
2      makeModel(prefixMapping);
3      var result = FrogChecks.checkFunctionExist.apply(model);
4      result = FrogChecks.checkVariableExists.apply(model).combine(result);
5      result = FrogChecks.checkGenericVariableExists.apply(model).combine(result);
6      result = FrogChecks.checkUnusedParameters.apply(model).combine(result);
7      result = FrogChecks.checkUnusedGenericParameters.apply(model).combine(result);
8      if (messageSeverityError(result)) return result;
9
10     result = FrogChecks.checkCntArgs.apply(model).combine(result);
11     result = FrogChecks.checkCntArgsGeneric.apply(model).combine(result);
12     result = FrogChecks.checkFunctionArg.apply(model).combine(result);
13     if (messageSeverityError(result)) return result;
14
15     setFunctionRef();
16     return validateArgumentsGenericArgumentsAndReturnType().combine(result);
17 }
18
19 private MessageHandler validateArgumentsGenericArgumentsAndReturnType() {
20     var mesgs = new MessageHandler();
21     functions.forEach((__, function) -> {
22         function.validateArguments(mesgs, getAllFunctions());
23         function.validateGenericArguments(mesgs);
24         function.validateReturnType(mesgs);
25     });
26     return mesgs;
27 }
```

a `MessageHandler` containing messages produced by the validation. As described in Section 4.5, we have implemented the validation in phases. Suppose the `MessageHandler` contains any messages with severity error after a phase. In that case, the validation will be stopped, and the `MessageHandler` containing the error messages will be returned. In short, the `validationFunctions` method consist of four main parts: making the Jena model containing the functions in the RDF query syntax, executing the SPARQL queries over the Jena model, setting the reference to functions in the function bodies[22], and finally executing the Java validation implementation on the functions.

5.5 Evaluation

In Section 4.6, we concluded that Frog should use lazy evaluation to evaluate function calls. For Lutra's Frog implementation to be lazy, we implement a `Map` as a lookup table to store a function's previous evaluations. Additionally, for the implementation to be non-strict, we utilise Java's `Supplier` interface[23]. This section discusses the implementation of memoisation and execution of function calls.

5.5.1 Memoisation

Before discussing our implementation of memoisation, we need to introduce two new concepts: a *function call signature* and a *function call description*. Firstly, a function call signature is its function name combined with its arguments. Secondly, the function call de-

[22] For instance replacing an IRI referring to a function with that function.

[23] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Supplier.html`

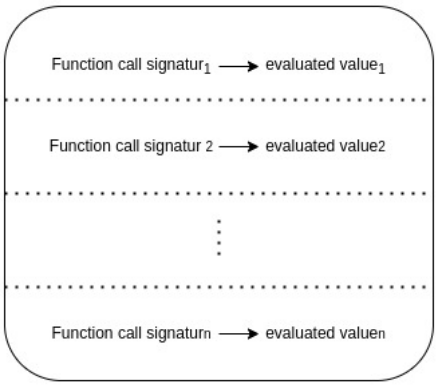

Figure 5.15: A generalisation of the lookup table when only considering the function call signature.

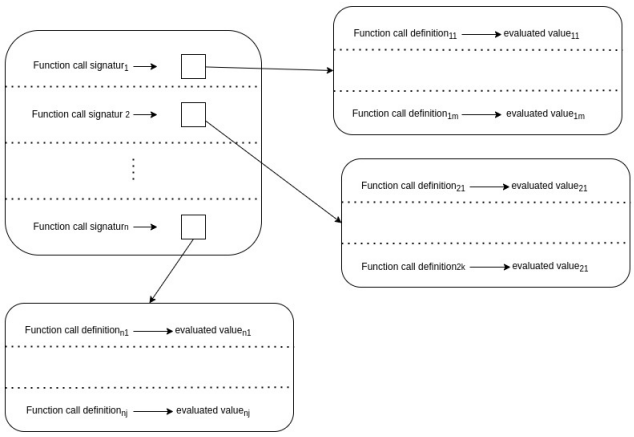

Figure 5.16: A generlaisation of the lookup table consdiering the function call signature and definition.

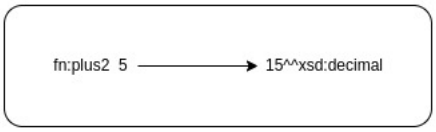

Figure 5.17: Example lookup table only considering the function call signature.

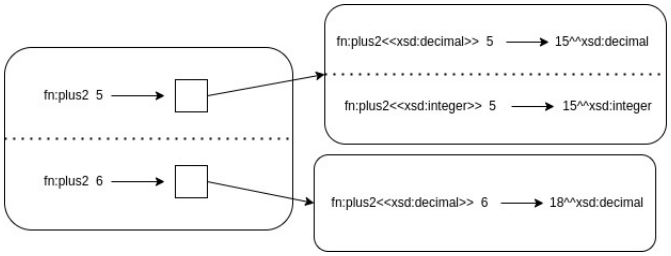

Figure 5.18: Example lookup table considering the function call signature and defintion.

scription is its function name, generic arguments and arguments. Thus the difference between the signature and description is that the description contains the generic arguments while the signature does not. Hence the function call `(fn:plus<<xsd:decimal>> 5 6)` and `(fn:plus<<xsd:integer>> 5 6)` have the same function call signature but difference function call descriptions.

Two function calls with the same function call signature always evaluate to the same value because Lutra performs the calculations over the same set of arguments. However, the return value's return type may vary due to the generic arguments. For instance, the function `ex:plus2` defined in Figure 4.13 can have different return types beacause the return type is determined from its generic parameter, `?T`. Thus the function call `(ex:plus2<<xsd :decimal>> 5)` evaluates to `15^^xsd:decimal` while `(ex:plus2<<xsd:integer>> 5)` evaluates to `15^^xsd:integer`. Therefore, utilising only the function call signature may result in wrong typing. For example, in the scenario of Figure 5.17 where Lutra has executed `(ex:plus2<<xsd:decimal>> 5)`, executing `(ex:plus2<<xsd:integer>> 5)` with that lookup table would result in a value with a wrong type, namely `xsd:decimal` instead of `xsd:integer`. Thus, a lookup table containing the function call signature and the evaluated value will not be enough.

Consequently, we implement a solution in Lutra, which first checks if the function call signature is present. If the signature is not present, the function call is evaluated and

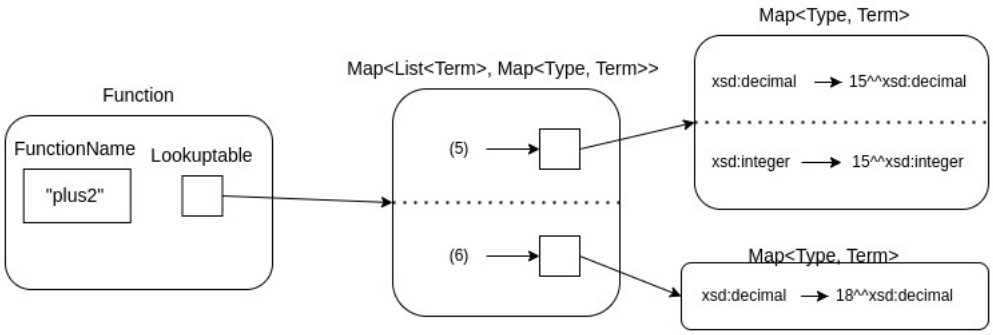

Figure 5.19: An illustaration of the lookuptable with Java types.

appended to the lookup table. However, if the signature is present, we utilise another table that maps function call descriptions to the evaluated value with the correct type. The value is extracted if the function call description exists in the second table. On the other hand, if the function call description is not present, we extract the value from another function call description and recalculate the value's type. Thus, we only evaluate each function call signature once and recalculate the type if necessary. The recalculation of the evaluated value's type will only be calculated once per possible return type. Figure 5.16 depicts the structure generally, while Figure 5.18 shows the instance containing the aforementioned function calls.

We implement the structure mentioned above by extending the `Function` class to contain a `Map` named `lookupTable` where the key represents the function signature. However, since the map is inside the Function class, the function name is stated implicitly. Thus, in practice, the `lookupTable`'s key is a list of terms referring to the arguments. Moreover, the value in the `lookupTable` is another map. In the inner map, we could have had the full function call description; however, in practice, we are only interested in the return type. Consequently, the second map has a type as its key and the evaluated value with the correct type as the value. As a result, different function call descriptions with the same function call signature and return type will, in practice, use the identical entry in the lookup table. The lookup table has the following Java type `Map<List<Term>, Map<Type,Term>>`.Figure 5.19 shows how Figure 5.18 looks in practice.

5.5.2 Execution

To delay execution of the function call until the value is needed, we implement an `execute` method that returns a `Supplier`[24] that again returns a `Result` object containg a term. This `execute` method is a part of the `Term` interface. By default, this method returns a `Supplier`, which returns the result of itself. However, we have overridden this method in four classes: `FunctionCall`, `FunctionTerm`, `FrogFunction`, and `BaseFunction`. Moreover, this `execute` method takes in the values needed to substitute the functions as arguments, namely the arguments and generic arguments. As a result, the `execute` method has the following signature:

[24] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Supplier.html`

```
public Supplier<Result<Term>> execute(List<Term> arguments,
                                      List<GenericType> genericArguments)
```

In this section, we examine the implementation in the four classes that override the `execute` method. However, before discussing these implementations, we want to note one case where Frog evaluates a function call even though a base function does not require it, namely, if the function call's name is a function call. Formally, a function call name can only be a variable or an IRI referring to a known function. However, during substitution and due to lazy evaluation, a function call that evaluates to a function can replace a variable that is the function call's name, hence becoming the function call name. Figure 5.20 exemplifies this scenario.

```
#Function using a function variable as function name in the function call
def ex:ParameterFunctionName(Function<xsd:string, xsd:boolean> ?fun)
    -> xsd:boolean :: (
    ?fun "hello"
).

#A function call on the function
(ex:ParameterFunctionName
    (fn:head (ex:stringIsEmpty ex:stringIsNotEmpty))
)

# Substituting the function call arguments into ex:ParameterFunctionName

((fn:head (ex:stringIsEmpty ex:stringIsNotEmpty)) "hello")
```

Figure 5.20: A scenario where the function call's name is a function call after substitution due to lazy evaluation.

FunctionCall

The `execute` method in the `FunctionCall` class consists of calling on the `execute` method on the function call's name with its arguments and generic arguments as the argument. If the function call's name is a function call, then the method evaluates this function call before calling on the `execute` method. Hence, the function call's name is either a Frog function, base function or function term when the method calls on the `execute` method. The `FunctionCall` class returns the result returned when calling on the `execute` method.

FunctionTerm

The `FunctionCall` class's `execute` method sends its arguments and generic arguments as arguments on its name's `execute` method. If this name is a `FunctionTerm` object, we reach the implementation of the `execute` method in the `FunctionTerm` class. A function term only consists of a reference to a function, a base or Frog function, and a set of generic arguments. In short, the `execute` method in the `FunctionTerm` calls on

the `execute` method on its function sending in the arguments it received and its generic arguments. Consequently, we replace the generic arguments sent in by a function call with the function terms generic arguments.

FrogFunction

The `FrogFunction` class's `execute` method has two primary tasks. Firstly, substitute the arguments and generic arguments into the function body and secondly, to preserve lazy evaluation as the evaluation strategy. In order to substitute the function body, we create a clone of the function's body where every variable is replaced with its corresponding argument or generic argument, as seen on lines 5-10 in Code 5.5.1. It is essential to create and substitute a clone of the function body rather than substitute the function body since a substitution with different arguments and generic arguments result in different function bodies.

To preserve lazy evaluation as the evaluation strategy, we implement the `FrogFunction` class' `execute` method in two parts: the first part before and outside the Supplier (on lines 4 to 11 in Code 5.5.1) and the second part inside the Supplier (on lines 13 to 20 in Code 5.5.1). The part before the Supplier performs the substitution and calls the `execute` method on the result of the substitution, resulting in a Supplier object. We delay the execution of this Supplier object until the second part, inside the Supplier. Consequently, the implementation recursively substitutes and creates Suppliers; however, these Suppliers are not executed before needed.

Code 5.5.1. *The execute method in the FrogFunction class.*

```
1  @Override
2  public Supplier<Result<Term>> execute(List<Term> arguments,
3                                        List<GenericType> genericArguments){
4    if(getLookupTable().containsKey(arguments)){
5      return () -> getResultFromLookupTable(arguments, genericArguments);
6    }
7    var substitutedFunctionBody = functionCall.substitute(
8                                  makeParToArgMap(arguments),
9                                  makeGenericMap(genericArguments));
10   var execution = substitutedFunctionBody.execute(List.of(),List.of());
11   return () -> {
12     if(getLookupTable().containsKey(arguments)){
13       return getResultFromLookupTable(arguments, genericArguments);
14     }
15     var result = functionCallExecution.get();
16     var returnType = execution(genericArguments);
17     if(result.isPresent())  result.get().setType(returnType);
18     addResultToLookupTable(arguments, result, returnType);
19     return result;
20   };
21 }
```

Additionally, we implement the memoisation and lookup table as elaborated in Section 5.5.1. As seen in Code 5.5.1, we perform a lookup in the table twice in this `execute` method, on lines 4 and 13. The two lookups in the method manage two different scenarios. Firstly, the code utilises the lookup on line 4 if we have calculated the function for the given arguments in a previous function call. For instance, when evaluating the `(ex:ParameterFunctionName ex:stringIsEmpty)` twice. This lookup removes unnecessary substitutions and creations of suppliers since the code already contains the result

```
ex:plus3<<?T subtypeOf owl:real>>(?T ?number1,?T ?number2) -> ?T :: (
    fn:plus<<?T>> ?number1 ?number2
).

#FUNCTION CALL
(ex:plus3<<xsd:integer>> (ex:plus2<<xsd:integer>> 5) (ex:plus2<<xsd:integer>> 5))
```

Figure 5.21: An example of where the second lookup will be used in Code 5.5.1.

of executing this function with the set of arguments[25]. Secondly, the code utilises the second lookup if the same function call signature occurs more than once while evaluating a function call. Figure 5.21 illustrates a scenario where the code applies the second lookup. In this case of Figure 5.21, the Supplier to the first `ex:plus2` will be evaluated before the second one. However, the evaluation of the first `ex:plus2` occurs after creation of the second Supplier. Consequently, the second `ex:plus2`'s can not utilise the first lookup, it can, however, use the second lookup.

BaseFunction

Finally, we discuss the implementation of the `execute` method in the `BaseFunction` class. As informed in Section 4.1.1, a base function performs a single task on a set of values, in other words, a set of arguments. The base functions are divided into two different groups, applying different technologies: The base functions that work on base types applies Saxon's XPath API[26], while the base functions working on OTTR specific types and the if function, known as the special functions, is implemented with pure Java. Figure 5.22 illustrates the general flow of the `BaseFunction` class's execute method.

In difference to Frog Functions, the base functions can create error messages. Lutra produces these error messages if the execution of the base function for a particular set of arguments is illegal but impossible to detect before evaluation. Examples of cases where the base function creates error messages are if we try to divide a number by zero or try to retrieve the first element of an empty list. Moreover, similar to the `FrogFunction` class the `BaseFunction` class also implements memoisation. The `BaseFunction` class's `execute` method performs memoisation in two cases:

1. Start of the supplier, check if the function call signature is previously applied[27].

2. After the execution of the arguments since executing the arguments may change the function call signature. For instance, the function call `(fn:plus<<xsd:integer>> (fn:plus<<xsd:integer>> 1 1)1)` signature changes after evaluating the terms because we replace the first argument with 2. Consequently, the function signature `(fn:plus (fn:plus 1 1) 1)` changes to `(fn:plus 2 1)`.

[25]The method `getResultFromLookupTable` handles the case where the lookup table only contains function call signature.

[26]`https://www.saxonica.com/documentation11/documentation.xml`

[27]The BaseFunction does not perform substitution since they do not contain a function body.

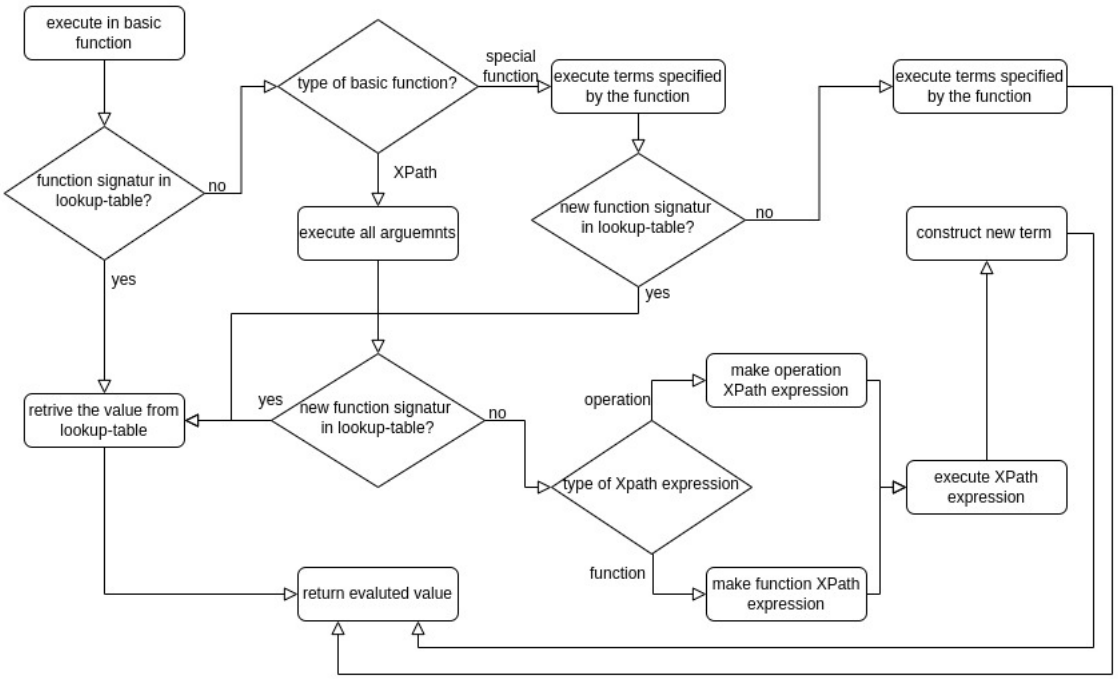

Figure 5.22: A flow diagram over the implementation in the BasicFunction class in Lutra. Does not include the producing of possible Messages.

We apply Saxon's XPath API to execute the base functions based on XPath functions and operations. We chosen to use Saxon's XPath API because the built-in XPath library in Java only supports XPath 1.0[28]. However, we see a need for applying functions and operations implemented in the later version of XPath, such as the functions for finding the smallest or highest number. Saxon's XPath API supports XPath version 2.0 and 3.1[29] and has an open source home edition.

In short, our XPath implementation consists of two steps: firstly, to convert the function or operation into an XPath expression and secondly, to evaluate this XPath expression. Lutra builds up an XPath expression in two ways, depending on whether the base function refers to an operation or a function. A base function based on a operation builds its XPath expression by placing the operation between the terms: term (operator term)*. For instance, Lutra transforms the function call `(fn:plus<<xsd:integer>> 2 1)` into the XPath expression 2 + 1. On the other hand, a base function that refers to a XPath function creates the following pattern: functionIRI'(' (term (',' term)*)* ')'. Thus, Lutra transforms the function call `(fn:concat "he""llo")`, which utilises the XPath function `xpf:concat`, into the XPath expression xpf:concat("he", "llo"). XPath can not interpret Frog's function call, thus, Lutra needs to evaluate the function calls before creating the XPath expression. For instance the function call `(fn:plus<<xsd:integer>> (fn:plus <<xsd:integer>> 1 1)1)` can not be written into `(fn:plus<<xsd:integer>> 1)+ 1, as (fn:plus<<xsd:integer>> 1)` is not a valid term in XPath. Consequently, all XPath base functions require Lutra to evaluate the arguments.

[28] `https://www.w3.org/TR/1999/REC-xpath-19991116/`
[29] `https://www.w3.org/TR/xpath-functions-31/`

Furthermore, implement the special function by creating pure Java functions. We refer to these Java functions as special functions. The special functions operates on the OTTR specific types, such a the list types. Lutra implements its list terms by using the `List` interface. Consequently, the special function regarding list uses the methods defined in Java's `List` interface. The list interface can not be applied to a function call term. Therefore if a list is a function call term, the function call term is evaluated. Another special function is the if-function. The if-function returns either the first or the second argument based on the third argument. Consequently, the if-function requires Lutra to evaluate the first argument and the second or third based on the first evaluated value.

5.6 Integrating Frog Functions with OTTR Templates in Lutra

This section discusses our changes in Lutra's OTTR implementation. We discuss how we validate function calls and function terms inside a template body or instance. Since we in Definition 4.5.4 expanded the Definition of a valid template dataset to include that every template and instance has term correctness. Moreover, we discuss our changes in Lutra's expansion after the discussion in Section 4.6.2.

5.6.1 Validating function terms utilised in templates

Lutra's OTTR template must validate that every function term used as an argument in an instance is correct after Definition 4.1.5. To validate that function terms has correctness, we need to validate the three points described in Section 4.5.1 regarding the function term.

As mentioned in Section 5.1, Lutra validates templates by applying queries in Lutra's query language. As previously discussed, when validating Frog Functions, we have chosen to apply SPARQL and pure Java methods implemented in the classes affected by the validation. However, validating the template function terms through the aforementioned methods would have required us to produce a RDF query syntax for template's heads and bodies and implement the constructions of this syntax of templates. Furthermore, validation on templates have already been implemented into Lutra; there exist several query methods previously created that we can reuse when constructing the three validations needed for the function term. Consequently, we have chosen to use Lutra's query language when validating function terms.

The construction of the function queries mainly consists of predefined queries; however, there was a need to implement seven queries specific for the function terms, the function type, and function, as shown in Table 5.1 and Table 5.2. Code 5.6.1 shows the validation made for validating that every argument used as a function exists, containg both the query and the error message produced when an validation error occurs.

Code 5.6.1. *The code used to validate that a term used as function exists. The new queryes are marked with the comment //new.*

```
1 private static final Check undefinedFunction = new Check(
2         Query.template("Temp")
3             .and(Query.bodyInstance("Temp", "Ins"))
```

```
4               .and(Query.argumentIndex("Ins", "Index", "Arg"))
5               .and(Query.usedAsType("Ins", "Index", "Lvl", "UsedAs"))
6               .and(Query.containsFunctionType("UsedAs")) //new
7               .and(Query.hasOccurenceAt("Arg", "Lvl", "Term"))
8               .and(Query.not(Query.functionExist("Term"))), //new
9           tup -> Message.error("Function not found error in template "+tup.get("Temp") + ": "
10                  + tup.getAs(Instance.class, "Ins").getIri()
11                  + " expects function(s) as arguments on index "
12                  + tup.get("Index") + ", however " + tup.get("Term")+" is not an function."
13          )
14      );
```

Name	Job	Parameters	Returns
functinExists	Checks whether an IRI can be connected to a base function or a function in the function library.	1. A string, which should be bound to an IRI in the tuple.	A stream of the tuple if the function exists and an empty stream otherwise.
getFunction	Retrieves a function based on the IRI.	1. A string that is bound to the function IRI. 2. A string in which the query binds to the function	The stream of the tuple containing the binding of the function object to the function string parameter.
getGenericParameters	Retrieves the generic parameter list to a function.	1. A string that is bound to the function. 2. A string in which the query bind the funtion's parameter list.	The stream of the tuple containing the binding of the parameter list object to the given parameter list string.
genericParameterIndex	Either retrieves the parameter on a given index, or create a stream for each parameter in the list.	1. A string that is bound to the parameter list. 2. A string that may or may not be bound to an index. 3. A string in which the query binds the parameter on the given index.	If the index is bound, the query returns the stream containing the binding of the parameter on the index to the given parameter string. Otherwise, the query returns a stream for each parameter in the list. Binding both the index and the parameter to tuple contained by the returned stream.

Table 5.1: The new queries introduced, working on Frog functions.

5.6.2 Validating function call terms utilised in templates

Similarly to the validation performed on the function calls in the function body, Lutra needs to validate that the function calls used in instances in the template body. Validating the function calls in the template body and function body is similar because they both need to validate that the function calls are correct. Section 4.5.1 contains the five validations needed for validating the correctness of function calls.

Implementing these validation with SPARQL only require appending the template heads of the templates containing function calls as arguments to the graph. The SPARQL queries demand the template heads to be present in the graph since queries need information about the type of the template head's parameters. Consequently, we only needed to append the template's head to the RDF query syntax. Additionally, the graph only need to contain the templates where function calls occur in the template body. In contrast, validating the function terms would need every template in the graph, as checking

Name	Job	Parameters	Returns
getGenericArguments	Retrieve the generic argument list from a function term. If the term is an IRI term, we create an empty list.	1. A string that is bound to the term. 2. A string in which the query bind the term's argument list.	the stream of the tuple containing the binding of the argument list object to the given argument list string.
genericArgumentIndex	Either retrieves the arguemnt on a given index, or create a stream for each arguemnt in the list.	1. A string that is bound to the arguemnt list. 2. A string that may or may not be bound to an index. 3. A string in which the query binds the argument on the given index.	If the index is bound, the query returns the stream containing the binding of the argument on the index to the given parameter string. Otherwise, the query returns a stream for each parameter in the list. Binding both the index and the argument to tuple contained by the returned stream.
containsFunctionType	Checks whether a type contains a the function type.	1. A string that is bound to the type.	A stream of the tuple if the type contains a functionc type, an empty stream otherwise.

Table 5.2: The new queries introduced, working on the function term and the function type.

whether we should interpret an IRI term as a function or an IRI depends on the corresponding parameter's type. Additionally, and most importantly, validating the function terms requires that we append the template body to the RDF query syntax as well as the template head. As a result of the reasons mentioned above, we decide to utilise SPARQL mainly due to two reasons: firstly, the validation mainly consists of SPARQL queries, which are already defined and therefore can be reused, and secondly, validating the function calls do not require us to define the whole template in the RDF query syntax only the template heads. Moreover, creating the queries in Lutra's query language would have needed many more defined queries regarding functions and function calls than when validating function terms.

To validate typing, correct subtype of arguments and generic arguments, we reuse the `validateArguments` and `validateGenrics` method in the `FunctionCall` class, as described in Section 5.4.3. However, we commit minor changes in the SPARQL queries. Firstly, by appending a separation between the function calls in a function body and template body in the RDF query syntax. We have separated the function call by establishing a triple connecting a blank node with the function calls used in a template with the predicate `frog:executableFunctionCall`. Additionally, this blank node relates to the IRI of the template containing the function call through the `frog:usedInTemplate` predicate.

Secondly, as Figure 5.23 illustrates, a function call in a template body can contain parameter variables defined in the template head. As seen in the queries from Section 5.4.2, the SPARQL queries need to know the type of the variable. For instance, Query 5.4.1 on lines 16-22 filters out every function call name defined as a parameter of type function in the function head. Consequently, we need to append the signature of the templates that has function calls in their body. The signature for template heads in the RDF query syntax is built up similarly to the function heads. The benefit of utilising the same structure is that we can reuse more extensive parts of the queries.

```
ex:functionTemplate[Function<xsd:integer,xsd:integer> ?fun] :: {
    ottr:Triple([], ex:functionTemplateExample, (?fun 1))
}.
```

Figure 5.23: An example template with containg a function call that utlises a parameter variable defined in the template head as the function call name.

Lastly, due to the additions to the RDF query syntax described above, the SPARQL queries' first step in the object queries and the arity queries are slightly different compared to the SPARQL queries used to validate functions. The difference is a result of that function calls in the function validation SPARQL queries are found through functions, while the function calls in template validation SPARQL queries are found through the predicate `frog:executableFunctionCall`. Query 5.6.1 depicts the SPARQL validation query for validating that the function name in the function call exists for function calls used in a template body. When comparing Query 5.6.1 with the counterpart Query 5.4.1, we can see that the difference in the query pattern is in the first part, namely the part that finds the function calls.

Query 5.6.1. *The validation query used to extract the function calls that does not use an exisiting function (either from the function library or a base function) or a parameter of the function type.*

```
1  SELECT ?functionCallName ?templateName
2  WHERE{
3      #finds everything used as a function call
4      #as argument in an instance
5      [] frog:executableFunctionCall/
6         (frog:arg/frog:val)*/frog:of ?functionCallName.
7
8      #removes all mathces where the function call name
9      #is an IRI and the IRI is of type function
10     FILTER NOT EXISTS{
11         ?functionCallName a frog:Function.
12         FILTER isIRI(?functionCallName)
13     }
14
15     #removes all mathces where a blank node is used as function is
16     #defined as a parameter of type function in the template
17     FILTER NOT EXISTS{
18         ?templateName a ottr:Template;
19                 frog:parameter [frog:var ?functionCallName;
20                                 frog:parameterType/a frog:Function]
21         FILTER isBlank(?functionCallName)
22     }
23 }
```

5.6.3 Expanding an instance/template containing function calls

As specified in Section 4.6.2, Lutra should evaluate the function calls as soon as they occur. However, if an instance contains a `None` value and a function call and the corresponding parameter to the `None` argument is not optional or contains a default value, then we have decided not to evaluate the function call. Since Lutra discards this instance regardless of the function call's evaluated value. Therefore we expand the `expandInstances` method in Lutra, depicted in Code 5.6.2 with the evaluation of function calls.

Code 5.6.2. *The pseudocode for expanding instances in Lutra with the addition of Frog.*

```
1  FUNCTION expandInstances(instance)
2      template <-gets the template with the same iri as the instance
3      IF template do not exist THEN
4          RETURN error
5      IF instance contain none at a non-optional posistion THEN
6          RETURN discard the instance
7      IF instance contains function call(s) THEN
8          RETURN instance with evaluated function call(s)
9      IF instance iri is a base template with no expander
10     OR the instace has ha expander but cannot expand THEN
11         RETURN instace
12     IF instance has list expandet THEN
13         generate instances, one instance per combination of the operator
14         RETURN expandInstaces on all the generated instances
15     ELSE
16         substitute the template's body with the instance's arguments
17         RETURN expandInstances all the substitute instances
```

Chapter 6

Discussion

In this chapter, we firstly discuss interesting matters on the design and implementation in Section 6.1. In Section 6.2, we evaluate whether the inclusion of Frog has improved OTTR's benefits of the DRY principle, better abstraction, uniform modelling, and separation of design and content. We introduce two case studies regarding producing RDF graphs from different sources that we solve using both OTTR without Frog and Frog. When discussing if Frog improves OTTR's benefits, we draw in examples from the two case studies and, from these examples, we discuss these benefits in a more general case.

6.1 Design & implementation

In this section, we discuss interesting topics of discussion that have occurd during designing and implementing Frog. We discuss why using SPARQL for validation instead the Lutra's query language is beneficial, the advantages and disadvantages of introducing a new syntax for querying and finally, how the addition of Frog conflicts with OTTR's quality of guarantee termination on a valid template library. The discussion regarding SPARQL for validation is an implementation discussion, while termination is a design discussion. The introduction of the RDF query syntax is both a design and an implementation discussion.

6.1.1 SPARQL and validation

Section 5.4 described our validation implementation for Frog functions in Lutra. As presented by that section, Lutra performed this validation by applying two different technologies, SPARQL and pure Java. Lutra, however, uses a custom created query language, written in Java, to validate OTTR templates and instances, as described in Section 5.1. Thus, maintaining Lutra's template and instance validation requires a maintainer to acquire knowledge on this specific means of querying. The OTTR project intends to replace this custom query language with a more established technology[1]. In this section, we argue why an established technology such as SPARQL is beneficial compared to Lutra's custom created query language. Additionally, we look into the limitations of SPARQL and how these limitations have resulted in the need for another technology to perform validation on types.

[1] After discussion with the OTTR team.

Before we discuss the proposed benefits of applying the established technology SPARQL rather than Lutra's query language, we need to identify why SPARQL is a well-established technology in regards to OTTR. Section 3.2 introduces SPARQL and describes SPARQL as a query language over linked data structured by RDF. Mosser et al. [30, p. 1] and Peréz er al. [33, p. 1] additionally state that SPARQL is the standard language to perform queries and extract data from RDF documents. W3C further reinforces this statement by endorsing SPARQL as a W3C reccomendation [42]. Hence, we argue that SPARQL is a well-established semantic technology. Furthermore, we assume that template programmers and maintainers of OTTR are knowledgeable about well-established semantic technologies due to OTTR being a macro language for the semantic technologies RDF and OWL. Consequently, we consider SPARQL a known and established technology for OTTR users and maintainers.

Now that we have introduced SPARQL as an established semantic technology in the context of OTTR, we proceed with arguing why we believe that conducting validation through SPARQL queries is beneficial compared to Lutra's query language. Firstly, applying SPARQL queries results in Frog validation leveraging the existing W3C stack. In addition, the SPARQL validation queries formally describe and specify Frog's validation. Similarly to how SHACL shapes and OWL vocabularies define OTTR's wOTTR serialisation and Frog's RDF serialisation. Lutra's query language, however, is not established outside the context of Lutra and does not have a specification that formally describes the language. Thus, we cannot utilise the queries written as a formal description of the validation. Secondly, the template programmers can understand the validation performed by Lutra straight from the SPARQL queries due to SPARQL being a well known technology.

Thirdly, we argue that applying a well-established technology, such as SPARQL, is easier to maintain rather than the Lutra specific query language. We assume that SPARQL queries are easier to maintain because template programmers can suggest and perform improvements as they are familiar with SPARQL. Additionally, with Lutra's current SPARQL execution implementation, one does not need to change the Lutra code[2]. Consequently, updating the validations later only require a programmer to know about SPARQL queries and the RDF query serialisation in isolation. Lastly, other implementations of OTTR can reuse the queries for validation. Thus, the SPARQL queries can work as a uniform means of validation across implementations. Utilising the queries made for Lutra query language across several implementations is more difficult as the query language is implemented to work in the context of Lutra and not other systems.

Performing all the validations in SPARQL has proven impracticable due to the incapacity to conduct recursive queries with SPARQL language [39, p. 711]. Recursive queries are necessary to perform queries concerned with typing because OTTR's lists and function types are nested. Reuter et al. discuss recursion in SPARQL and propose a recursive linear operator, which they implemented as an extension of Jena [39, p. 732]. However, implementing this solution in Lutra is out of scope for this thesis. Moreover, we made an effort to create a representation of the types in the RDF query syntax and SPARQL queries with property paths for the validations regarding types. However, in our findings

[2]Unless the changes require changes in the extraction of data from the query and the formulation of messages.

and to the best of our knowledge, there are no satisfactory solutions to this problem by applying pure SPARQL.

The optimal solution for type validation would be to find another semantic technology that makes it possible to recursively validate the typing through querying or another method. However, we were not able to find such a technology. Thus, we were limited to either constructing queries with Lutra's query language or validating through pure Java implementations. A motivation for utilising Lutra's query language would be to reuse and arrange the established constructions to validate typing in the Frog functions. However, Lutra does not currently have support for Frog-constructs. Hence, we would need to create a set of base relations in the Lutra query language for all of Frog's constructs. We argue that creating these base relations in the Lutra query language would be more complicated and complex than writing them in pure Java since Lutra already offers methods for type validation[3].. Moreover, the SPARQL validations validate that every value needed for the type validations are present[4]. Consequently, the Java validation implementations can assume that the values are present. This assumption allows us to write the Java implementation without null-pointer checks, making the code shorter and concise. Hence, we argue that a Java implementation is more readable and simpler to comprehend for a Frog function programmer compared to the queries produced with Lutra's query language. Thus, we believe that a pure Java validation implementation preserves the maintenance benefit of utilising SPARQL better than creating queries for Lutra's query langue. For the reason above, we chose to create a pure java implementation to validate the typing in the Frog functions, as seen in Section 5.4.

To summarise, we have leveraged SPARQL queries to the extent as technically possible and used relatively simple pure java methods where applying SPARQL is proven to be complicated. We argue that the benefits mentioned regarding SPARQL, such as getting a formal definition, reuse and maintenance, are more substantial compared to Lutra's query language; although we only leverage in parts of the validation.

6.1.2 RDF query syntax

As presented in Section 5.4.2, we have introduced a new syntax to Frog, the RDF query syntax. However, Frog already had an RDF syntax that SPARQL could have queried over before introducing the RDF query syntax. This section discusses why we chose to include another RDF syntax in the Lutra implementation instead of the already established RDF syntax.

Firstly we establish the possible benefits of using the existing RDF syntax. In the previous section, we discussed that one of the benefits of applying SPARQL rather than Lutra's query language is that it is easier to maintain. In addition, the SPARQL validation queries can be used as a formal description of the validation. However, introducing a new syntax compared to utilising an existing one would require comprehending the RDF query syntax in addition to the established RDF syntax. Additionally, using the established RDF syntax opens up the possibility to leverage even further of the W3C

[3]Section 5.4.3 elaborates on the methods `isSubtypeOf` and `isCompatibleWith`

[4]Due to the arity validation performed by SPARQL.

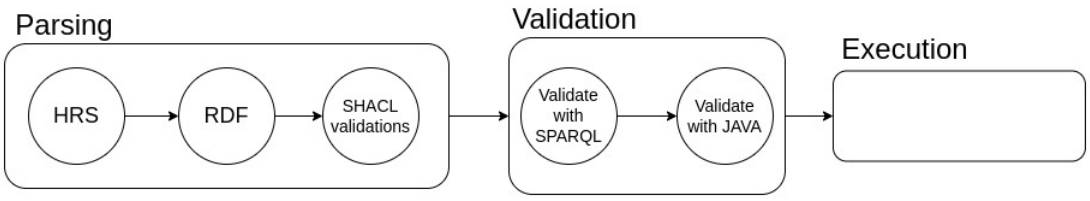

Figure 6.1: An alternative basic flow of a Frog implemntation.

stack. As illustrated by Figure 6.1, when utilising the established RDF syntax, we can translate the HRS syntax into the RDF syntax and then perform validations on the RDF graph with the SHACL shapes before validating directly on the graph with SPARQL. Consequently, we can leverage the W3C stack as far as possible before producing and working on Java objects. Additionally, several OTTR and Frog implementations can reuse this pipeline.

Another solution could be to introduce the RDF query syntax as a official Frog syntax, either by replacing the already established syntax or by having two different RDF syntaxes. Using the RDF query syntax as an official Frog syntax would require more validation. Since replacing list structures with blank node structures removes order and continuity regarding the parameters and arguments; because each blank node must state their index explicitly. Example 6.1.1 shows that the RDF query syntax allows us to write a function with parameters only on indexes 2 and 6. Consequently, if we were to introduce the RDF query syntax as an official Frog syntax, we would need to validate that the indexes regarding parameters and arguments form a proper list. In other words, that the indexes are in the range of 0 to N - 1, where N is the number of parameters or arguments.

Example 6.1.1. *An example showing that it is possible to write indexes in the RDF queries that are discontinuous.*

```
ex:plus2Times a frog:Function;
    frog:parameter [frog:type xsd:integer;
                    frog:var _:number1
                    frog:index 2],
                   [frog:type xsd:integer;
                    frog:var _:number2
                    frog:index 6].
```

The disadvantages of constructing queries over Frog's RDF syntax are a lack of data and metadata, primarily due to Frog's RDF syntax using lists to represent several constructs. As seen in query 6.2, many of the properties in the queries is property paths combining `rdf:rest` and `rdf:first`, for instance, finding the generic arguments of a function call, as seen on lines 10 to 12. In comparison, the RDF query syntax can use the property `frog:typeArg` to find blank nodes containing information about the generic arguments of a function call. When constructing queries for the two syntaxes, we experienced that the queries for the RDF syntax required additional subqueries to extract metadata. For example, finding the index of a parameter or an argument. In comparison, the RDF

```
SELECT *
WHERE {
  #FINDS EVERY FUNCTION CALL
  ?functionName a :Function;
                :def [].

  { #Used as/in a generic argument
        ?functionName :def/rdf:rest/
                rdf:rest/rdf:first/
                (rdf:rest*/rdf:first)*
                                ?functionCall.
        ?functionCall rdf:first :functionCall.

         ?functionCall rdf:rest/rdf:rest
                /rdf:first _:genericList.
          _:genericList rdf:first :typeArgs.
          _:genericList rdf:rest+/rdf:first/
                  (rdf:rest*/rdf:first)*
                          ?genericVariable.

  } UNION { #Used as/in the return type
        ?functionName :type/:returnType/
                (rdf:rest*/rdf:first)*
                          ?genericVariable.
  } UNION { #Used as/in parameter types
        ?functionName :type/:parameterTypes/
                rdf:rest*/rdf:first/
                (rdf:rest*/rdf:first)*
                          ?genericVariable.
  }

  FILTER(isBlank(?genericVariable))
  FILTER NOT EXISTS {
        ?functionName :typeVars/rdf:rest*/
                rdf:first/:var
                      ?genericVariable
  }
}
```

```
SELECT ?functionName ?genericVariable
WHERE {
     ?functionName a :Function;
                   :body [].

     { #Used as/in a generic argument
     ?functionName :body/(:arg/:val)*/:typeArg/
                   (:type/(:argType+/:type)+)?
                                  ?genericArgument
     } UNION { #Used as/in the return type
     ?functionName :returnType/
                  (:argType+/:type)* ?genericArgument
     } UNION { #Used as/in parameter types
     ?functionName :parameter/:parameterType/
                   (:argType+/:type)* ?genericArgument
     }

     ?genericArgument a :GenericType;
                      :type ?genericVariable.
     FILTER NOT EXISTS{
     ?functionName :typeVar/:var ?genericVariable.
     }

}
```

Figure 6.2: The query on the left is written for Frog's RDF syntax, while the query on the right is for the RDF query syntax. The two queries both finds undefined generic arguments in the function body.

query syntax contains metadata through the property `frog:index` about the indexes of the arguments and parameters. Thus, we argue that the readability of the queries for the RDF query syntax is better than Frog's RDF syntax. Moreover, the RDF query syntax contains information regarding the OTTR type of a term. As discussed in Section 6.1.1, we do not use queries to validate typing. However, if we were to construct these recursive queries[5], the RDF query syntax would be the most suitable solution.

Including yet another syntax has its disadvantages as well, especially in regards to maintenance. If we introduce another construct to Frog, we need to define the construct not only to the two official serialisations but also to the RDF query serialisation. Moreover, the translation from the two other syntaxes to the RDF query syntax introduces yet another step in implementing Frog. Hence, introducing the possibility of more bugs in the implementation. Frog would have required a translation between the HRS and RDF syntax if Frog used the RDF syntax to query over. However, a translation between the HRS and RDF syntax would be beneficial not only in regards to validation but also because HRS is the preferred syntax for writing functions and the RDF syntax for publishing functions. Consequently, with a translation, we can write the functions in HRS and get a Frog implementation to translate the functions to the RDF syntax for publication and visa versa.

To conclude, the choice of whether to make a new RDF syntax for querying or utilising Frog's existing RDF syntax is a choice between improved readability or not having to

[5]For instance, by appending Reutter et al.'s recursive clause [39] on top of Jena.

learn a new syntax. We chose improved readability because we argue that the complexity of understanding the queries for Frog's RDF syntax is higher compared to learning the RDF query syntax, even though this requires more maintenance in Frog implementations. An alternative solution can be to construct queries for both syntaxes; however, an effort to construct all the queries for Frog's RDF syntax was not made, in this thesis, due to time limitations[6].

6.1.3 Termination

An expansion of an instance on a template from a valid template library guarantees termination. This guaranteed termination stems from the definition of a valid template library, which the mOTTR specification [21] defines. Notably, the mOTTR specification defines that a valid template library must be acyclic[7]. In other words, a valid template library does not support recursion on templates, consequently guaranteeing termination.

However, Frog is a programming language allowing recursive functions, thus making it possible to create non-terminating functions. Example 6.1.2 illustrates a Frog function that never terminates when the input is a non-empty list. Finding out whether function calls terminate or not is impossible to validate before executing them[8], in contrast to validating that a template library is acyclic. Thus, we can not guarantee that a template which depends on a Frog function terminates, even though the template is a part of a valid template library. Consequently, when introducing Frog into OTTR, we remove OTTR's quality of guaranteed termination of an instance on a template from a valid template library.

Example 6.1.2. *An example of a Frog function that does not terminate if ?lst is a non-empty list.*

```
ex:addAll(List<xsd:integer> ?lst) -> xsd:integer :: (
    if<<xsd:integer>> (fn:isEmpty ?lst)
    0
    (fn:plus<<xsd:integer>> (fn:head<<xsd:integer>> ?lst) (ex:addAll ?lst))
).
```

A solution to the problem mentioned above could be to restrict Frog such that the functions in the function library need to have an acyclic dependence relation, similar to OTTR. Then Frog could offer the known terminating recursive functions map, filter and reduce as built-in Frog functions, making it possible to create recursive functions by applying them.

6.2 Improving OTTR by including Frog

To discuss if the addition of Frog actually has improved the OTTR framework, we look into two case studies. These studies compares a solution with and without Frog. Fur-

[6] Writing the queries for both RDF syntaxes would also require more maintenance.
[7] Definition 3.4.2 defines an acyclic template library.
[8] The Halting problem.

Table 6.1: An extraction of the CSV data retrieved from Extrasolar Planets Encyclopaedia. The actual CSV files contain more columns but have been removed in this extracti for simplicity.

name	planet_status	mass	star_name
11 Com b	Confirmed		11 Com
11 Oph b	Confirmed	21.0	Oph 1622-2405
14 And b	Confirmed		14 And

Table 6.2: An extraction of the execl data retrieved from NASA Exoplanet Archive. The actual excel files contain more columns but have been removed in this extraction for simplicity.

rowid	pl_hostname	pl_name	pl_bmassj
1	11 Com	11 Com b	19.4
2	11 UMi	11 UMi b	14.74
3	14 And	14 And b	4.8

thermore, we discuss which of OTTR benefits Frog strengthens in general terms based on examples from the two case studies. We also look into other notable benefits and disadvantages when including Frog into OTTR.

6.2.1 Case Study: Planets

Case Description and the Data Sources

Let us assume we have two data sets on planets from two different sources: the Extrasolar Planets Encyclopaedia[9] and the NASA Exoplanet Archive[10]. Data extraction from the Extrasolar Planets Encyclopaedia results in a CSV file. This CSV file contains data regarding planets with their name, planet status, mass and the name of the star it orbits, as seen in table 6.1. On the other hand, extracting data from the NASA Exoplanet Archive gives us an Excel table with data about a planet's name, mass, and the name of the star the planet orbits. Table 6.2 is an example of an extracted excel file from NASA.

We want to create a model of planets in RDF from these two sources. The following vocabulary should represent our model of a planet:

- The name of the planet.
- The relation from a planet to the star it orbits.
- The name of the star that the planet orbits, if present.
- The mass of the planet, if present.

Additionally, we want both the planets and the stars to have an IRI where their name is the local name. Figure 6.3 illustrates a general graph of how we desire to model a planet[11].

[9] `http://exoplanet.eu/catalog/`

[10] `https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=PS`

[11] This case is inspired by a case presented in a lecture in the subject IN5800 at the University of Oslo

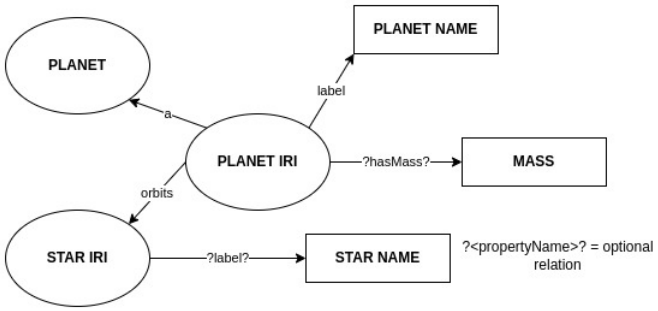

Figure 6.3: A generalisation of the structure of a planet in a RDF graph.

To create the RDF graph, we use OTTR. We solve this case both with and without Frog. Moreover, we utilise both bOTTR and tabOTTR to create instances from the files due to the retrieved data being in a CSV and an Excel format.

Without Frog

When working with OTTR without Frog, all the terms and values must be calculated before creating the instances. Consequently, the template modelling a planet must, among other things, take in the IRIs of the planets and stars as arguments. As previously mentioned, these IRIs need to include the planet's or star's name as their local name. Thus, we have created Template 6.2.1, which encapsulates our modelling of a planet.

Template 6.2.1. *The template modelling a planet without the use of Frog.*

```
ex:Planet [ :IRI ?iri, xsd:string ?name, :IRI ?star,
    ? xsd:string ?starName, xsd:decimal ?mass] :: {
    o-rdf:Type(?iri, ex:Planet),
    o-rdfs:Label(?iri, ?name),
    :Triple(?iri, ex:orbitsStar, ?star),
    o-rdfs:Label(?star, ?starName),
    ottr:Triple(?iri, ex:hasMass, ?mass)
} .
```

As seen in table 6.1 and 6.2, neither of the sources contain IRIs for the planets and stars. Thus, we need to create the IRIs through the mapping or excel sheet with the tabOTTR preamble. The planet's and star's names, which our IRIs are based on, may contain spaces in their name. Spaces are not allowed in IRIs. We replace spaces with an underscore (_) in our mapping and excel sheet. As seen in Figure 6.4, our mapping removes the spaces with the `REPLACE` function and combines the localname and name of the planet or star with the `CONCAT` function; resulting in a valid IRI. We produce the IRIs in the Excel sheet by creating two new columns containing the IRIs, which Excel calculated through the following formula:

```
=CONCAT("http://example.xyz/ns/",
      SUBSTITUTE(<pl_hostname or pl_name cell>, " ", "_"))
```

Table 6.3 shows the result of adding the tabOTTR preamble and the calculated columns to table 6.2.

`https://leifhka.org/in5800/lectures/mappings/slides/index.html` .

```
ex:Planet a :InstanceMap ;
  :template ex:Planet ;
  :query
  """SELECT
    CONCAT('http://example.xyz/ns/', REPLACE(name, ' ', '_')),
    name,
    CONCAT('http://example.xyz/ns/', REPLACE(star_name, ' ', '_')),
    star_name,
    mass
    FROM CSVREAD('<csv file path>',null,'charset=UTF-8 fieldSeparator=,');""" ;
  :argumentMaps ( [ :type :IRI ] [ :type xsd:string ]
                  [ :type :IRI ] [ :type xsd:string ]
                  [ :type xsd:decimal ]) ;
  :source [ a :H2Source ] .
```

Figure 6.4: Our mapping creating instances of ex:Planet from a CSV file with the format shown in table 6.1.

Table 6.3: Table 6.2 with tabOTTR preamble and the calculated IRIs.

#OTTR ex #OTTR #OTTR	prefix *http* : *//example.xyz/ns/* end template	 ex:Planet			
0	4	2	1	3	5
	xsd:string	xsd:string	iri	iri	xsd:decimal
rowid	pl_hostname	pl_name	pl_iri	pl_hostiri	pl_bmassj
1	11 Com	11 Com b	ex:11_Com	ex:11_Com_b	19.4
2	11 UMi	11 UMi b	ex:11_UMi	ex:11_UMi_b	14.74
3	14 And	14 And b	ex:14_And	ex:14_And_b	4.8
#OTTR	end				

```
def ex:toIRIWNamespace(xsd:string ?localName) -> ottr:IRI :: (
    ex:toIRI "http://example.org/data/" ?localName
).

def ex:toIRI(xsd:string ?namespace, xsd:string ?localName) -> ottr:IRI
:: (
    fn:castToIRI (fn:concat ?namespace (fn:translate ?localName " " "_"))
).
```

Figure 6.5: Frog function's used in the template to generate IRIs in the template.

With Frog

In contrast to the solution without Frog, the addition of Frog makes it possible to create functions that can manipulate terms inside templates. Consequently, as Figure 6.5 illustrates, we can create Frog functions that creates an IRI based on a namespace and a local name. We have created the function `ex:toIRIWNamespace` since all of the instances have the same namespace[12]. Then we can use this Frog function in the template to calculate the IRI based on the names of the planets and stars. As seen in Template 6.2.2, we have chosen to create a new Template `ex:PlanetFrog`, with a body containing an instance of `ex:Planet` from Template 6.2.1 with the necessary function calls to create IRIs.

Template 6.2.2. *The template modelling a planet with the use of the Frog function, which constructs IRIs.*

```
ex:PlanetFrog [xsd:string ?name,? xsd:string ?starName,? xsd:decimal ?mass]
:: {
    ex:Planet((ex:toIRIWNamespace ?name),?name,
              (ex:toIRIWNamespace ?starName), ?starName, ?mass
             )
} .
```

As a consequence of the template performing the calculations of IRIs, the mapping and excel with tabOTTR preamble does not need to perform or contain any calculation, only extracting of terms to create instances, as seen in Figure 6.6 and table 6.4.

6.2.2 Case Study: Weather stations

Case Description and the Data Sources

This case is inspired by the motivation example regarding weather stations introduced in Section 1.1. We have extended this case further, including more data to model. In short, we want to model historical data on weather stations in RDF. A weather station should contain the average temperature, maximum temperature, minimum temperature, amount of rain, and snow depth for each date if present. Additionally, a weather station should contain the name of its location, and the station id should be the local name of

[12] In the subsequent case study, we see that this removes repetition.

```
ex:Planet a :InstanceMap ;
  :template ex:PlanetFrog ;
  :query
  """SELECT name, star_name, mass
   FROM CSVREAD('<CSV file path',null,'charset=UTF-8 fieldSeparator=,');""" ;
  :argumentMaps ([:type xsd:string] [:type xsd:string][:type xsd:decimal]) ;
  :source [ a :H2Source ] .
```

Figure 6.6: Our mapping creating instances of ex:PlanetFrog from a CSV file with the format shown in table 6.1.

Table 6.4: Table 6.2 with tabOTTR preamble.

#OTTR ex #OTTR #OTTR	prefix *http* : *//example.xyz/ns/* end template	 ex:Planet	
0	2	1	3
	xsd:string	xsd:string	xsd:decimal
rowid	pl_hostname	pl_name	pl_bmassj
1 2 3	11 Com 11 UMi 14 And	11 Com b 11 UMi b 14 And b	19.4 14.74 4.8
#OTTR	end		

the station's IRI. Furthermore, the model should store the temperature measurements in Celcius (C) and the rain and snow depth measurements in millimetres (mm). Figure 6.7 illustrates how we want to model the weather stations in a general graph.

We have retrieved data from two different sources: Natural Centers for Environmental Information[13] (USA) and from Meteorlogisk institute[14] (Norway). We extracted five CSV files from the Centers for Environmental Information in the same format containing data for five weather stations placed in different cities across the USA[15]. The first line of the CSV file contained a string with the station's name and its id, for instance, `"LAGUARDIA AIRPORT, NY US (USW00014732)"`. We removed this data from the CSV file and placed the information directly into the mapping for each file. Additionally, we have shortened some of the column names, for example, *TAVG (Degree Fahrenheit)* to *TAVG*. Table 6.5 illustrates the format of the CVS files after this cleaning. These CVS files contain their temperature data in Fahrenheit (F) and their data regarding snow depth and rain in inches (in).

Moreover, we retrieved data from three weather stations in different cities[16] in Norway from Meteorlogisk institute in an excel format. The data from Meteorlogisk institute

[13] `https://www.ncei.noaa.gov/access/past-weather/`
[14] `https://seklima.met.no/observations/`
[15] More specific: New York, Boston, Orlando, Seattle and Los Angeles.
[16] More specific: Oslo, Trondheim, and Bergen.

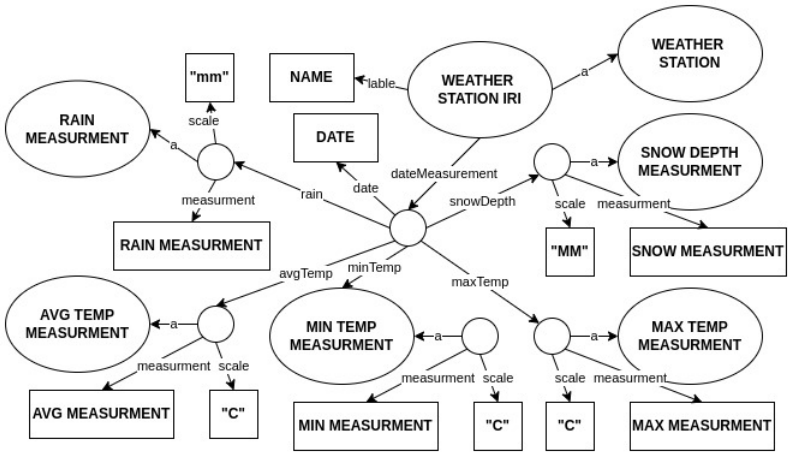

Figure 6.7: A generalisation of the structure of a weather station in the RDF graph.

Date	TAVG	TMAX	TMIN	PRCP	SNOW	SNWD
2011-09-01	78	83	63	0.00		0.0
2011-09-02	77	83	61		0.0	0.0
2011-09-03		86	59	0.00	0.0	0.0

Table 6.5: The table illustrates the format of the CSV data from Natural Centers for Environmental Information, after cleaning.

uses the scale Celcius (C) and millimetre (mm). The files represented no data with a cell with only a dash character (-); however, we replaced these cells with empty cells for OTTR to interpret the cells as a none existing value. Additionally, we have shortened the column names in the excel sheets; the column *Tid(norsk normaltid)* has, for instance, been shortened down to *Tid*. Table 6.6 shows a generalisation of the format of the excel sheets retrieved from Meteorlogisk institute with our cleaning regarding empty cells.

Without Frog

As mentioned in the case description, the model should only contain Celcius (C) and millimetre (mm) data. Consequently, when using OTTR without Frog, we assume that the data in different scales are calculated into Celcius (C) and millimetre (mm) before creating the instances. Thus, we constructed Template 6.2.3, which encapsulates our model from the case description.

Furthermore, we have created one mapping for each of the CSV files extracted from the Natural Centers for Environmental Information. However, as previously explained,

Navn	Stasjon	Tid	Maktemp	Midtemp	Mintemp	Snodybde	Nedbor
Bergen	SN50540	02.01.2010	-2.7	-8.3	-9.9	20	0.1
Bergen	SN50540	03.01.2010	-2.9	-5	-10	20	0
Bergen	SN50540	04.01.2010	-0.1		-4.2	22	1.1

Table 6.6: The table illustrates the format of the excel data from Meteorlogisk institute, after cleaning.

Template 6.2.3. *The template modelling a weather stations without the use of Frog.*

```
ex:WeatherStationCelcius[:IRI ?iri, xsd:string ?name, xsd:date ?date, ? xsd:decimal ?avgTemp,
                    ? xsd:decimal ?minTemp, ? xsd:decimal ?maxTemp, ? xsd:decimal ?snow, ? xsd:decimal ?rain]
:: {
    o-rdf:Type(?iri, ex:WeatherStation),
    o-rdfs:Label(?iri, ?name),
    :Triple(?iri, ex:dateMeasurement, _:measurement),
    :Triple(_:measurement, ex:date, ?date),
    ex:Measurement(_:measurement, ex:avgTemp, ex:AvgTempMeasurment, ?avgTemp, "C"),
    ex:Measurement(_:measurement, ex:avgTemp, ex:MinTempMeasurment, ?minTemp, "C"),
    ex:Measurement(_:measurement, ex:avgTemp, ex:MaxTempMeasurment, ?maxTemp, "C"),
    ex:Measurement(_:measurement, ex:rain, ex:RainMeasurment, ?rain, "mm"),
    ex:Measurement(_:measurement, ex:snowDepth, ex:SnowDepthMeasurment, ?snow, "mm")
} .

ex:Measurement[:IRI ?iri, ! :IRI ?prop, :IRI ?type, xsd:decimal ?meas, xsd:string ?scale] :: {
    :Triple(?iri, ?prop, _:measurment),
    o-rdf:Type(_:measurment, ?type),
    :Triple(_:measurment, ex:measurment, ?meas),
    :Triple(_:measurment, ex:scale, ?scale)
} .
```

```
ex:MapBosten a :InstanceMap ;
  :template ex:WeatherStationCelcius ;
  :source [a :H2Source];
  :query """SELECT
              \'http://example.xyz/ns/USW00014739\',
              \'BOSTON, MA US\',
              CAST(Date as date),
              ROUND((CAST(TAVG as decimal) - 32) * 5/9, 1),
              ROUND((CAST(TMIN as decimal) - 32) * 5/9, 1),
              ROUND((CAST(TMAX as decimal) - 32) * 5/9, 1),
              ROUND(CAST(SNWD as decimal) * 25.4, 1),
              ROUND(CAST(PRCP as decimal) * 25.4, 1),
          FROM CSVREAD('<path to boston csv file>', null, 'charset=UTF-8 fieldSeparator=,');""" ;
  :argumentMaps (
    [ :type :IRI ] [ :type xsd:string] [ :type xsd:date ] [ :type xsd:decimal ]
    [ :type xsd:decimal ] [ :type xsd:decimal ] [ :type xsd:decimal ][ :type xsd:decimal ]
    ) .
```

Figure 6.8: The map creating instances of `ex:WeatherStationCelcius` *from a CSV file with the format shown in table 6.5*

the data in the CSV files use Fahrenheit (F) and inches (in) as scales. Therefore, each mapping needs to convert the measurements into the correct scale. We have created the conversion using numerical operation, as seen in Figure 6.8. The excel files from Meteorlogisk institute, on the other hand, represent their measurement data in Celcius (C) and millimetre (mm); consequently, no conversion is needed. However, we need to create IRIs based on the Stasjon column for the weather stations. The creation of the IRIs uses the same formula as the case study regarding planets in the previous section. Table 6.7a shows the result of adding the IRIs to the excel sheet shown in table 6.2 and the tabOTTR preamble.

With Frog

As seen in the section above, the mappings and the excel files have different types of calculations. However, it is possible to move these calculations from mappings and excel files into the template using the functions defined in Figures 6.9 and 6.5, as seen in Template 6.2.4. Template 6.2.3 assumes that the terms used as arguments in instances are on the correct scale. When we move these calculations into the templates, we can no longer assume that the measurements are on the correct scale. Therefore, in contrast to Template 6.2.3, Template 6.2.4 takes in the scale of temperatures, snow depth, and rain scale. Template 6.2.4 applies `ex:convertToMMIfInches` and `ex:convertToCIfF` to check whether the measurements are on the correct scale and convert them if necessary.

Table 6.7: Tables for the weather case. Without Frog on the left and with Frog on the right.

(a) Table 6.2 with tabOTTR preamble and the calculated IRIs. For the solution without Frog.

#OTTR ex #OTTR #OTTR	prefix *http://example.xyz/ns/* end template	ex:WeatherStationCelcius						
2	0	1	3	4	5	6	7	8
xsd:string		iri	xsd:date	xsd:decimal	xsd:decimal	xsd:decimal	xsd:decimal	xsd:decimal
Navn	Stasjon	StasjonId	Tid	Maktemp	Midtemp	Mintemp	Snodybde	Nedbor
Bergen	SN50540	ex:SN50540	02.01.2010	-2.7	-8.3	-9.9	20	0.1
Bergen	SN50540	ex:SN50540	03.01.2010	-2.9,-5	-10	20	0	
Bergen	SN50540	ex:SN50540	04.01.2010	-0.1		-4.2	22	1.1
#OTTR	end							

(b) Table 6.2 with tabOTTR preamble and columns for the scales. For the solution with Frog.

#OTTR ex #OTTR #OTTR	prefix *http://example.xyz/ns/* end template	ex:WeatherStationCelcius							
2	1	3	4	5	6	7	8	9	10
xsd:string	xsd:string	xsd:date	xsd:decimal	xsd:decimal	xsd:decimal	xsd:decimal	xsd:decimal	xsd:string	xsd:string
Navn	Stasjon	Tid	Maktemp	Midtemp	Mintemp	Snodybde	Nedbor		
Bergen	SN50540	02.01.2010	-2.7	-8.3	-9.9	20	0.1		
Bergen	SN50540	03.01.2010	-2.9,-5	-10	20	0			
Bergen	SN50540	04.01.2010	-0.1		-4.2	22	1.1		
#OTTR	end								

```
def ex:inchesToMM(xsd:decimal ?inces) -> xsd:decimal :: (
    fn:times<<xsd:decimal>> ?inces 25.4
).

def ex:convertToMMIfInches(xsd:decimal ?number, xsd:string ?scale) -> xsd:decimal :: (
    fn:if<<xsd:decimal>> (ex:stringEquals ?scale "I")
        (ex:inchesToMM ?number)
        ?number
).

def ex:convertToCIfF(xsd:decimal ?number, xsd:string ?scale) -> xsd:decimal :: (
    fn:if<<xsd:decimal>> (ex:stringEquals ?scale "F")
        (ex:FtoC ?number)
        ?number
).

def ex:getFinalScale(xsd:string ?scale) -> xsd:string :: (
    fn:if<<xsd:string>> (fn:or (ex:stringEquals ?scale "F") (ex:stringEquals ?scale "C"))
        "C"
        "MM"
) .

def ex:stringEquals(xsd:string ?str1, xsd:string ?str2) -> xsd:boolean :: (
    fn:if<<xsd:boolean>> (fn:equal<<xsd:integer>> 0 (fn:compare ?str1 ?str2))
        true
        false
).
```

Figure 6.9: The functions needed for Template 6.2.4. Figure 4.7 defined the function `ex:FtoC` *which function* `ex:convertToCIfF` *utilises.*

Template 6.2.4. *The template modelling a weather stations with the use of Frog.*

```
ex:WeatherStationCelcius[ottr:string ?stationID, xsd:string ?name, xsd:date ?date, ? xsd:decimal ?avgTemp, ? xsd:decimal ?minTemp,
                         ? xsd:decimal ?maxTemp, ? xsd:decimal ?snow, ? xsd:decimal ?rain,
                         xsd:string ?tempScale = "C", xsd:string ?snowAndRainScale = "mm"] :: {
    o-rdf:Type((ex:toIRIWNamespace ?stationID), ex:WeatherStation),
    o-rdfs:Label((ex:toIRIWNamespace ?stationID), ?name),
    ottr:Triple((ex:toIRIWNamespace ?stationID), ex:dateMeasurement, _:measurement),
    ottr:Triple(_:measurement, ex:date, ?date),
    ex:Measurement(_:measurement, ex:avgTemp, ex:AvgTempMeasurment, ?avgTemp, ?tempScale, ex:convertToCIfF),
    ex:Measurement(_:measurement, ex:avgTemp, ex:MinTempMeasurment, ?minTemp, ?tempScale, ex:convertToCIfF),
    ex:Measurement(_:measurement, ex:avgTemp, ex:MaxTempMeasurment, ?maxTemp, ?tempScale, ex:convertToCIfF),
    ex:Measurement(_:measurement, ex:rain, ex:RainMeasurment, ?rain, ?snowAndRainScale, ex:convertToMMIfInches),
    ex:Measurement(_:measurement, ex:snowDepth, ex:SnowDepthMeasurment, ?snow, ?snowAndRainScale, ex:convertToMMIfInches)
} .

ex:Measurement[ottr:IRI ?iri, ! ottr:IRI ?prop, ottr:IRI ?type, xsd:decimal ?meas,
               xsd:string ?scale, Function<xsd:decimal, xsd:string, xsd:decimal> ?fun] :: {
    ottr:Triple(?iri, ?prop, _:measurment),
    o-rdf:Type(_:measurment, ?type),
    ottr:Triple(_:measurment, ex:measurment, (fn:roundPrecision<<xsd:decimal>> (?fun ?meas ?scale) 1)),
    ottr:Triple(_:measurment, ex:scale, (ex:getFinalScale ?scale))
} .
```

```
ex:MapBosten a :InstanceMap ;
  :template ex:WeatherStationCelcius ;
  :source [a :H2Source];
  :query """SELECT
              'USW00014739', 'BOSTON, MA US', Date,
              TAVG, TMIN, TMAX, SNWD, PRCP,'F','in'
          FROM CSVREAD('path to boston csv file', null, 'charset=UTF-8 fieldSeparator=,');""" ;
  :argumentMaps (
    [ :type xsd:string ] [ :type xsd:string] [ :type xsd:date ]
    [ :type xsd:decimal ] [ :type xsd:decimal ] [ :type xsd:decimal ] [ :type xsd:decimal ] [ :type xsd:decimal ]
    [ :type xsd:string ] [ :type xsd:string ]
    ) .
```

Figure 6.10: The map creating instances of `ex:WeatherStationCelcius` *from a CSV file with the format shown in table 6.5.*

As a consequence of moving the calculation of measurements into the templates in Template 6.2.4, the template `ex:WeatherStationCelcius` needs to know the scales of the measurements. Therefore, we have added data regarding the scales in the mappings and Excel files, as seen in tables 6.10 and 6.7b. Note that sources with millimetre and Celcius as their scales do not explicitly need to state their scales due to the default values in Template 6.2.4[17]. Moreover, by moving the calculation of IRI into the templates, we can remove the calculation to produce IRIs in both the mappings and the Excel files.

6.2.3 Discussion

Don't repeat yourself (DRY)

In Section 1.1, we argued that Frog would improve the *Don't repeat yourself* (DRY) principle. We claimed that by moving calculations from mappings or through tools for handling tabular data, such as Excel, and into the templates, we only needed to add one procedure instead of one for each source type. The planet case presented in Section 6.2.1 is an example of how we can take advantage of Frog to remove repetitious calculations. In the planet case, when we utilised OTTR without Frog, we produced two different computation methods for the same task, namely to produce IRIs for the stars and planets with their name as the local name. In contrast, OTTR with Frog only required one Frog function to calculate the IRIs, namely `ex:toIRIWNamespace`. Thus, applying Frog functions instead of calculations in the mappings and tabular files makes it possible to remove unnecessary repetitions of similar calculations.

In general, different sources have different approaches for calculating the same values. For instance, the binding `BIND(?IRI, CONCAT(?namespace, REPLACE(?localName, " ", "_")))` could have been used to produce IRIs in a mapping with a SPARQL source. In contrast, PostgreSQL would similarly calculate the IRIs with the function call `concat(<namespace>, translate(<local name>, " ", "_"))`. As a result, when utilising OTTR without a Frog Function, one must create one method for the same calculation for each source type[18]. When utilising a Frog Function inside the template instead, only one calculation method is needed regardless of the number of different sources.

Moreover, the introduction of Frog has made it possible to remove repetitious calculation over the source type with the same structure, as seen in the weather station case presented

[17] In this case, this regards the Excel files, as seen in table 6.7b.

[18] Where the calculation is required.

in Section 6.2.2. In the weather case without Frog, we constructed five mappings, one for each CSV file, containing the same calculations for converting Fahrenheit to Celcius and inches to millimetres, as seen in Figure 6.8. When we utilised Frog, on the other hand, we only defined one function for each conversion which we used in Template 6.2.4. Only having one definition for each calculation, a Frog function, is beneficial as it is easier to maintain one function compared to N functions, where N is the numbers of mappings or tabular files.

If the calculated value is required in several places in the template, utilising Frog functions may result in the template containing repetitious function calls. Template 6.2.4, for the weather station case, is an example of repetitious function calls resulting from utilising Frog functions[19]. Changes to the IRI definition, for instance, what should be in the local name, would require a change in three different function calls in the template. In contrast, no changes would be necessary for Template 6.2.3, where Frog functions are absent. However, changing the local name would have required changing each mapping and tabular file. Consequently, both solutions, in this case, are not optimal with regards to the DRY principle. A solution, when utilising Frog, could have been to create a wrapper template as seen in the planet case with Template 6.2.2.

To conclude, through the planet and weather station case and the general case, we see that the inclusion of Frog has strengthened OTTR's benefit of the DRY principle. This principle is strengthened because OTTR with Frog only requires one Frog function while OTTR without Frog requires one calculation method for each source. Additionally, the inclusion of Frog facilitates the possibility to remove repetitious calculations in mappings and tabular sources of similar structure into one Frog function.

Better Abstraction & Uniform modelling

OTTR claims to provide better abstraction, as OTTR templates create an abstraction layer between data and model or structure of the data, consequently ensuring uniform modelling [29, p. 50]. For instance, in the planet case presented in Section 6.2.1, we created a model i.e. case description that, among other things, contained that a planet was related to the star it is orbiting. Additionally this model contains that the local name of the planet's and star's IRIs should be their name. Nevertheless, Template 6.2.1, which did not apply Frog functions, could not encapsulate the premises regarding the IRIs. As a result, the mappings and tabular files need to ensure that this part of our model is correct. On the other hand, Template 6.2.2, which applies Frog functions, encapsulated the model's description of the IRIs directly into the template. Consequently, we argue that the inclusion of Frog in OTTR has made it possible for the OTTR templates to model not only the structure of the data but also the logic on terms stated by the model. Including the model's logic on terms in the templates results in the template abstracting the logic on terms, hence improving uniform modelling.

The weather case introduced in Section 6.2.2 also emphasises the above claim. Particularly in regards to the scale of the measurements. As explicitly stated in this case description, the scale of the measurements should either be in Celcius or millimetres.

[19] The function calls in the first three instances calculating the IRI of the weather station.

Without Frog, we could not model these criteria directly into the template. Therefore, we needed to assume that the data were on the correct scale. With Frog, we moved the necessary conversions into the templates. Thus, this template became an abstraction not only for the structure and pattern but also for the logic over terms.

A benefit of these abstractions, as previously stated, is that they strengthen OTTR's claimed benefit of uniform modelling. As presented in the previous discussion regarding the DRY principle, without Frog, the mappings and tabular files need to perform the calculations before creating the instances. In contrast, with Frog, the templates can perform these calculations. As seen in the planet case, we needed, without Frog, to create one method for the mapping and one for the tabular file for the same calculation. The number of different methods increase as the number of different sources increase. However, when placing the calculation into the templates, we only use one Frog function regardless of the number of sources. Hence, this results in one uniform method for performing calculations on terms. Without Frog, a logical change in the model regarding the terms requires modifications in every method of calculations in the different mappings and tabular files. In contrast, utilising one uniform Frog function in the template would only require modifications in the Frog function. Thus, we argue that OTTR with Frog is easier to maintain then OTTR without Frog.

The possibility of placing the calculations in the templates can, in some cases, create templates that are easier to use, as we move the complexity from mappings and tabular files into templates. When comparing the mappings and tabular files in the two cases, we see that the mappings and tabular files used for templates with Frog functions are less complex than those without Frog. However, as seen in the weather station, some extra data regarding the measurements' scale was required by Template 6.2.4 due to using Frog functions. Notably, there is a difference between adding data regarding the scales and, for instance, adding IRIs. This difference is that we produce IRIs across sources, while adding data regarding scales only concerns the data within the specific mapping or tabular file. Moreover, we argue that adding the data regarding the scales in the mappings and tabular files is less complex than converting the degrees in the mappings.

Furthermore, when calculating terms inside the templates, we ensure that data follows the given model. As discussed in the weather case, Template 6.2.3 assumes that the measurements were on the correct scale. Consequently, if the data used to create the instances is on an incorrect scale and not converted, expanding the instances would create logical errors in regards to the model. However, Template 6.2.4, which performs the necessary conversions, ensures that our logic regarding the scales is correct.

Lastly, we note that the inclusion of Frog makes it possible to create all of the IRIs inside the template, which is an RDF or semantic technology-specific requirement for a resource, as seen in both cases. Hence, the OTTR templates can be an abstraction for RDF or semantic technology-specific elements.

To summarise this discussion, we believe that the addition of Frog strengthens OTTR's benefits of better abstraction and uniform modelling; since OTTR with Frog allows us to encapsulate a model's logic on terms inside the templates. As discussed, better abstrac-

tion and uniform modelling offer several benefits, such as ease of use and less maintenance.

Seperation of design and content

In the previous discussion, we argued why Frog improved the uniform modelling and abstraction by using the template as an abstraction for the structure of the data and the logic over terms. Following these discussions, we can also argue that Frog strengthens OTTR's benefit of separation of design and content. Regarding the separation of design and content, the OTTR team argues that the templates create a natural separation between the design of the knowledge base and the content [38]. In our two cases, we can refer to the model or case description as the knowledge base and the data extracted from the tabular files and mappings as the content since they create the instances. Placing Frog functions inside templates makes it possible to encapsulate the logic over terms. Hence, creating a further separation of design and content; as we move our design of logic over terms from the content (instances and creation of instances) and into the design (templates).

For instance, in the weather case study presented in Section 6.2.2, we needed to calculate temperatures, snow, and rain in mappings. These calculations were a part of our design. Thus, we placed parts of our design in the creation of instances, which represent the content. However, with Frog, we placed these calculations in the template, which represents the design. Consequently, we argue that the inclusion of Frog makes it possible to create an improved separate design and content.

Other notable points

The previous discussions in this section have focused on how the addition of Frog has improved some of OTTR's benefits, which is the focus of this thesis. However, we would like to introduce and shortly discuss some of the inconveniences and disadvantages of applying Frog compared to performing calculations in the mappings and tabular files.

In Section 1.2, we noted that the focus of this thesis was not to implement an efficient implementation of Frog. However, it is interesting to note that it is more time-consuming to produce an RDF graph for Lutra with Frog compared to Lutra without Frog. Figure 6.11 illustrates the time on the OTTR execution used to determine the efficiency of applying Frog compared to performing the calculations with SQL in the bOTTR mappings. This graph shows the time used by OTTR to expand N weather case instances. We performed the test on 100, 1000, 10 000 and 100 000 instances, each executed ten times, producing an average time to plot the graph[20]. The data were extracted from the Natural Centers for Environmental Information[21] and created the instance by applying the bOTTR mappings shown in Section 6.2.2.

Moreover, we need to compare how error-prone the two solutions are. As previously discussed, the addition of Frog makes it possible to create one uniform function compared to one for each source to compute the same calculations. Creating several methods for the same calculations is more error-prone than only having one function. On the other hand,

[20] Appendix C stores the complete data set on the time takings.

[21] `https://www.ncei.noaa.gov/access/past-weather/`

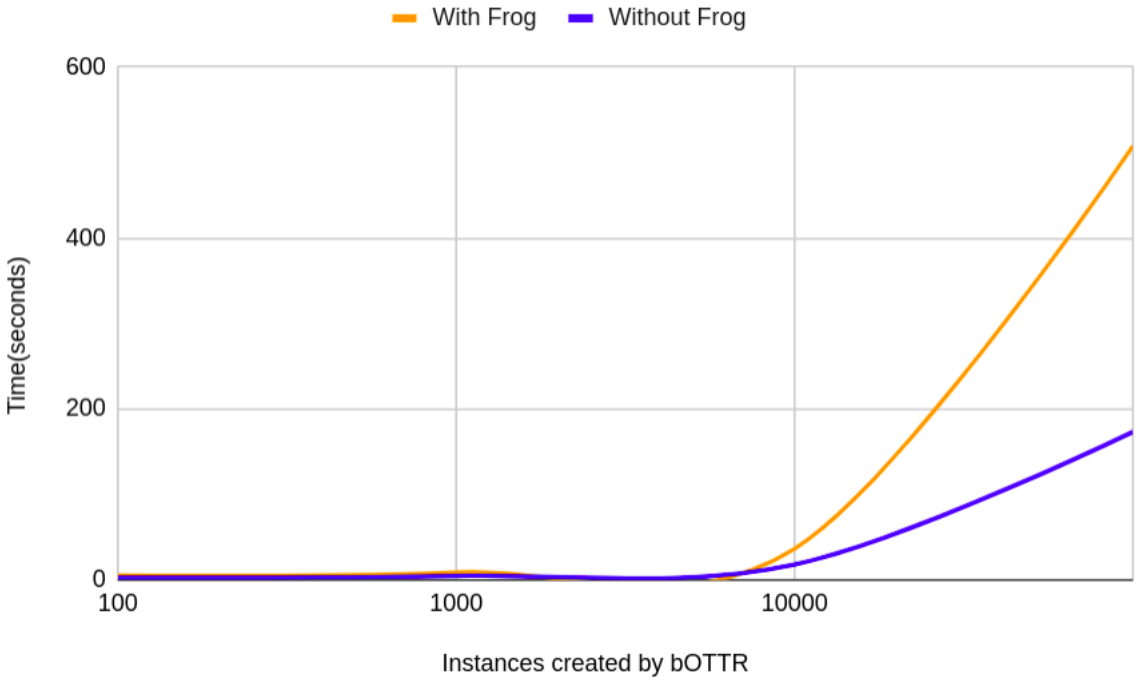

Figure 6.11: An graph showing the difference time it took to expand instances over OTTR when performing the calculation with Frog functions versus in the mappings. In other words, Frog function versus SQL functions.

compared to the technologies used for handling tabular files and extracting data from sources through bOTTR, Frog is more likely to contain errors in the source code, which may produce errors when performing the calculation. Due to the other technologies being well-established, hence, better tested than Frog. In conclusion, Frog is more error-prone when considering the code, while the use of Frog, on the contrary, is less error-prone compared to the technologies used with tabular files and mappings over bOTTR.

In Section 6.1.1, we discussed the benefit of using a well-established technology rather than creating and using a specific one. In this case, we can argue that creating Frog is using a specific way of performing calculations over terms rather than utilising the existing methods through the mappings and tabular files, which apply well-established technologies. Consequently, many developers of, for instance, bOTTR mappings already know how to create the necessary calculations in the technology for that source. Utilising Frog could then be considered yet another technology to learn. However, a notable difference between the discussion in Section 6.1.1 and this discussion is choosing between two solutions and adding another method to perform the calculations. Including Frog into OTTR does not remove the possibility of performing calculations in the mapping and tabular files; it only adds the possibility to perform the calculations inside the templates. Thus, learning and applying Frog to abstract a model's logic over terms into the templates is an option for the template programmer, not a requirement.

6.2.4 Summary of discussion and conclusions

In this chapter, from two case studies that use real-life data, we have seen how the inclusion of Frog is beneficial and how the inclusion of Frog has strengthened OTTR's benefits of the DRY principle, better abstraction, uniform modelling and separation of design and content. Moreover, we have discussed some of the disadvantages of including Frog, such as that utilising Frog function is more time-consuming than performing the calculations in mappings. Notably, the addition of Frog has made it possible to move calculations into the templates. However, the inclusion of Frog does not remove the possibility of conducting calculations in the mappings and tabular files. Consequently, the user of Frog can choose whether to perform the calculation in the templates or the mappings or tabular files. Nevertheless, we suggest, from the findings in the discussions, using Frog and templates to execute calculations instead of mappings and tabular files in the following cases:

1. The calculation is a part of the model or knowledge base, such as creating an IRI based on another parameter.

2. Calculations that are not a part of the model or knowledge base but are needed by several sources and tabular files. A wrapper template can perform these calculations.

Chapter 7

Related Work

The introduction of Frog into OTTR tries to improve some of the benefits by making it feasible to manipulate terms inside the templates. As presented in the previous chapter, we intentionally designed Frog such that it seamlessly integrates with OTTR. For instance, by choosing to use the same type and term system. Frog and OTTR having the same type of system is especially important because OTTR performs type validation on the arguments in the instances, as discussed in section 4.5. Importantly, Frog is also designed to be placed and executed inside the templates and not on the expanded graph.

In this chapter, we introduce alternative approaches to Frog with the focus on existing technologies that allow for manipulations of RDF terms. Since the task of this thesis is to integrate manipulation of terms into OTTR seamlessly, we will discuss the different technologies with a focus on the following set of criteria.

1. We require that the technology can be integrated into the templates and not work on a graph, as this would require the calculations to be performed after the expansion of the ground instances. Consequently, it would be impossible to manipulate and create terms that OTTR requires for the expansion, such as producing IRI to identify the current resource uniquely.
2. The technologies need to be statically typed[1] since OTTR requires type validation of the argument in instances.
3. Moreover, the type system must be compatible with the OTTR type system; this would, among other things, require the type system to have a compatible counterpart to the OTTR specific LUB and list types.
4. We require that the typing of the terms are compatible, for instance, that `"Hello"`, which OTTR types as an `xsd:string` is typed as an `xsd:string` by the technology. Thus the term system must be compatible, especially regarding the OTTR list term.
5. We require that there exists a publicly available implementation of the technology.
6. We will discuss if the technology has any known limitations

Table 7.1 summarises the discussion in this section based on the previously mentioned criteria.

[1] Or have an easy way to make it statically typed.

Table 7.1: Shows which criteria the different technologies mentioned in this section fullfiles

	Semantic Technologies	SHACL Functions	Ripper	Adenine
1. Manipulate terms during the expansion	X	✓	✓	X
2. Is statically typed	X	✓	X	X
3. Compatible type system with OTTR	X	X	X	X
4. Compatible term system with OTTR	X	X	X	X
5. Accessible (or open-source)	✓	✓	✓	X
6. Does not have too significant restrictions	✓	✓	X	?

7.1 Semantic Technologies

Several established semantic technologies offer manipulation of terms or values, such as SPARQL [42] and SWRL [14]. SPARQL, for instance, contain a set of functions, which is a subset of XQuery 1.0[2] and XPath 2.0 functions and operators[3] on terms, that can be utilised in SPARQL queries. A combination of INSERT queries with functions and DELETE queries makes it feasible to insert manipulated values into the graph and remove values that only were in the graph to calculate the manipulated values previously inserted. Moreover, SWRL, a proposed language for the semantic web, can express rules that can modify a graph by appending additional data based on the rules [14]. As with SPARQL, SWRL offers a function based on functions and operations from XPath and XQuery. Example 7.1.1 shows an example of a SWRL rule.

Example 7.1.1. *An example of a SWRL rule that calculate the degree in Celcius based on the Fahrenheit and insert it into the graph.*

```
ex:hasFahrenheitDegree(?x, ?fD)  ->
  ex:hasCelciusDegree(?x,
   swrlb:multiply(swrlb:substract(?fD, 32), swrlb:divide(5,9)))
```

These two semantic technologies work on and extract data from existing graphs, which is also the case for many other semantic technologies such as OWL[4]. Thus we can apply these technologies on a graph, however, not on a single term inside the templates. Consequently, when using semantic technologies after the expansion, we make it feasible to calculate terms inside the templates but after, hence removing the calculations from the OTTR system altogether. Furthermore, these technologies also come with their limitations. As noted, functions in SPARQL is built on a subset of XPATH 2.0 and SWRL on a subset of an unspecified version of XPath. Thus, we are limited to the functions and operations these technologies have chosen to utilise.

7.2 SHACL functions

In the previous section, we discussed semantic technologies in general through the examples of SWRL and SPARQL. However, we want to explicitly discuss an advanced feature

[2] https://www.w3.org/TR/2010/REC-xquery-20101214/
[3] https://www.w3.org/TR/xquery-operators/
[4] https://www.w3.org/TR/owl2-overview/

in SHACL, namely SHACL functions. SHACL Advanced Features working group note [2] introduces the SHACL functions as a work in progress; however, it is still an interesting technology to look at regarding calculations in OTTR. A SHACL function produces a single RDF term from an arbitrary number of parameters and is defined with an IRI uniquely identifying the function, similarly to Frog. The functions can be defined either from SPARQL [2] or a SHACL-js document [27], making it possible to point to an implemented JavaScript function. Moreover, we can state the parameter types and return types. However, this is not required; consequently, being dynamically typed. Thus, utilising SHACL functions would require us to validate that the parameter types and return types are stated in the SHACL functions; this validation is, however, fairly easy to implement. Example 7.2.1 is an example of a SHACL function with SPARQL that adds values together.

Example 7.2.1. *An example of a SHACL function applying SPARQL that adds to numbers together.*

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex:  <http://example.net/ns#> .
@prefix sh:  <http://www.w3.org/ns/shacl#>

ex:plus a sh:SPARQLFunction ;
    sh:parameter [
                    sh:path ex:num1;
                    sh:dataType xsd:integer;
                    sh:order 0
                ],
                [
                    sh:path ex:num2;
                    sh:dataType xsd:integer;
                    sh:order 1
                ];
    sh:returnType xsd:integer ;
    sh:select """
        SELECT ($num1 + $num2 AS ?res)
        WHERE {}
    """.
```

Notably, the following two reasons make SHACL functions hard to utilise in OTTR: firstly, there is no well-known public implementation[5]. For instance, the established semantic web API in java Apache Jena [3] does not implement SHACL functions, only SHACL core and SHACL SPARQL constraints. Furthermore, none of the public available implementations in the SHACL Test Suite and Implementation Report [10] implements SHACL functions. Secondly, the type system for SHACL functions is not the same as the OTTR type system. Notably, the SHACL type system uses the XSD schema for types such as xsd:integer, meaning that some of OTTR's basic types have a corresponding and compatible type in the SHACL function type system. There are, however, no corresponding types for the OTTR specific types, such as the Lists and LUB types. Consequently, it

[5]To the best of our knowledge.

would have been necessary to create a translation between the type systems and perform it in OTTR.

7.3 Ripple

Ripple is a functional stack-based scripting language made for manipulating on RDF graphs and RDF terms, with a syntax inspired by the turtle serialisation of RDF. Mainly, Ripple offers queries and commands or directives [43]. The directive @define can, combined with a query expression, produce a function, as seen in figure 7.1. Furthermore, Ripple offers a set of core libraries, such as the math and string library. We can find the complete library set in the publicly available implementation of Ripple[6].

```
# n => factorial(n)
@define fact:
    /dup 0 /equal # if n is 0...
    (1 /popd) # yield 1
    (/dup 1 /sub /:fact /mul) # otherwise, yield n*fact(n-1)
    /branch. # => 120

5/:fact.
```

Figure 7.1: Shows an function in Ripple that recursively adds together the factorial number. This example is taken from Shinaver's article [43, p. 6].

However, Ripple is not statically typed and consequently not compatible with the OTTR system. Furthermore, utilising Ripple would have restricted us to produce functions by combining the function in the Ripple libraries and Ripple queries. Thus, we need to depend that these libraries and the implementation are relatively regularly updated when shortages are found. However, the last release that we could locate was from 2016.

7.4 Adenine

The Haystack project introduced Adenine as an imperative scripting language to manipulate RDF-encode metadata [18, pp. 9-10]. One of the motivations behind Adenine was to create a language with a syntax that supported the RDF data model, which the Haystack team believed to remove unnecessary verbose compared to utilising programming languages like Python and C++ [18, pp. 9-10]. To perform the functions, Adenine extracts data from an RDF container performs operations on them and append new statements to the same RDF container [18, pp. 9-10][7].

Consequently, Adenine works on an RDF graph and not on a single RDF term, thus making it hard to utilise inside OTTR for the same reasons as with the semantic tech-

[6] `https://github.com/joshsh/ripple`

[7] Adenins interpreter is written in Java, and the developers have made it possible to access the call on Java methods [18, pp. 9-10].

nologies. Furthermore, Adenine is not a typed language, thus not having a compatible type system with OTTR.

7.5 Summary

To summarise, we have in this chapter discussed possible alternative approaches to Frog and shed light on existing technologies made for manipulating data in the semantic web. As discussed, several of these technologies work on RDF graphs and not single RDF terms, making it hard to incorporate them into OTTR templates. Furthermore, and most notably, none of the technologies presented in this chapter has a compatible term and type system with OTTR. Table 7.1 summarises the criteria the different technologies fulfil and do not fulfil.

We have not discussed a solution to use existing scripting languages, such as Python and Java. However, since OTTR has OTTR specific types, it would require a translation of the typing system. Moreover, Skjæveland et al. [44] have discussed several related works regarding OTTR relevant to Frog, such as Tawny OWL.

Chapter 8

Conclusion

In this thesis, we have designed and developed a functional programming language, Frog, that seamlessly integrates with OTTR, making it possible to modify and perform calculations over terms inside OTTR templates. By introducing Frog into the OTTR framework, we have improved OTTR's following benefits: Don't repeat yourself (DRY) principle, better abstraction, uniform modelling and separation of design and content.

Frog seamlessly integrates with OTTR as function calls on Frog functions have become a part of OTTR's term system. Hence, using a function call term as an argument in a template's body is as natural as placing an integer. Additionally, the inclusion of Frog preserves the semantics of OTTR, both in terms of validation of template datasets and expanding instances. Firstly, Frog preserves OTTR's validations by validating the Frog functions. Validating the Frog functions ensures that a function's return value's type corresponds with this function's stated type. Consequently, as the type of a function call is the affiliated function's stated return type, Frog guarantees that the function call terms have the correct type. Moreover, Frog and OTTR have the same term and type system. Hence, easing the use of OTTR because we do not need to be aware of a translation of terms and types when creating Frog functions. Secondly, we preserved OTTR's expansion semantics by evaluating function calls in an instance before expanding it. Thus, function calls work in terms of discarding instances and list expanders. Notably, OTTR without Frog guarantees termination on a valid template library. However, a Frog function does not guarantee termination as the functions can have cyclic dependency relations that may lead to infinite recursion. Thus, including Frog into OTTR removes OTTR's quality of termination.

The inclusion of Frog has enhanced OTTR's benefits, and is making it possible to further encapsulate our design of an RDF graph. OTTR without Frog encapsulated this design's structure. On the other hand, OTTR with Frog encapsulates this design's structre and logic over terms. In this paragraph, we refer to the design of an RDF graph as a data model. Firstly, Frog has improved the Don't repeat yourself (DRY) principle since we can create one Frog Function instead of one function for each data source when using OTTR to map data to an RDF graph, reducing repetition. Secondly, Frog has enhanced the better abstraction benefit and uniform modelling as templates can use Frog functions. With the possibility of applying frog functions in templates' bodies, we can now encapsulate both a data model's data structure and logic over terms. Additionally, making it possible to abstract logic over terms inside templates. Thirdly, we have strengthened OTTR's

benefit of separation of design and content by including Frog because the data model's logic over terms can now be encapsulated in the templates. Consequently, the mappings and tabular files, which extract or contain data, can solely focus on creating instances from the content and not on performing calculations on the content such that it follows the data model.

8.1 Future Work

We suggest the following points as interesting future work regarding Frog and OTTR:

- As discussed in Section 6.2.3, using Frog may result in repetitious function calls in a template body if the calculated value is needed as several arguments. A possible approach is binding of variables inside the template body.
- In this thesis, we did not focus on the efficiency of our Frog implementation. One may still choose to perform calculations in the mappings and tabular files since applying Frog functions is slower in comparison, especially when OTTR expands many instances. Having an efficient implementation of Frog will presumably make Frog a more suitable solution not only in theory but also in practice.
- In Section 6.1.1, we argued why we believe well-established semantic technologies are be beneficial for validating Frog functions. However, SPARQL has limitations regarding recursive querying, making it difficult to create queries over our encoding of types in RDF. A further investigation into how different semantic technologies may be used to perform these types of validations may be of interest, both in regards to Frog and OTTR.
- In Section 6.1.3, we discussed how the addition of Frog into OTTR has removed OTTR's quality of guaranteed termination. A solution to keep this quality is to offer a set of terminating higher-order functions as a basis to conduct iterations and to make all other Frog functions' dependency relation acyclic. We propose that for further work, one can inspect if the removal of guaranteed termination has a tangible impact on the OTTR framework and which changes in Frog are needed if the impact is significant.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. eng. MIT Electrical Engineering and Computer Science. Cambridge: The MIT Press, 1996. ISBN: 9780262510875.

[2] Dean Allemang, Simon Steyskal, and Holger Knublauch. *SHACL Advanced Features*. W3C Note. https://www.w3.org/TR/2017/NOTE-shacl-af-20170608/. W3C, June 2017.

[3] *Apache Jena SHACL*. URL: `https://jena.apache.org/documentation/shacl/index.html` (visited on 03/10/2022).

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web: A New Form of Web Content That is Meaningful to Computers Will Unleash a Revolution of New Possibilities". In: *ScientificAmerican.com* (May 2001), p. 1.

[5] Katalin Bimbo. *Combinatory logic : pure, applied and typed*. eng. Boca Raton, 2012.

[6] Gavin Carothers and Eric Prud'hommeaux. *RDF 1.1 Turtle*. W3C Recommendation. https://www.w3.org/TR/2014/REC-turtle-20140225/. W3C, Feb. 2014.

[7] Alonzo Church. *The calculi of lambda-conversion*. eng. Princeton, N.J, 1985.

[8] H. B. Curry. "Grundlagen der Kombinatorischen Logik". In: *American Journal of Mathematics* 52.3 (1930), pp. 509–536. ISSN: 00029327, 10806377. URL: `http://www.jstor.org/stable/2370619` (visited on 04/27/2022).

[9] Richard Cyganiak, Markus Lanthaler, and David Wood. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. W3C, Feb. 2014.

[10] Jose Emilio Labra Gayo, Holger Knublauch, and Dimitris Kontokostat. *SHACL Test Suite and Implementation Report*. Tech. rep. https://w3c.github.io/data-shapes/data-shapes-test-suite/. W3C, Jan. 2021.

[11] Jose Emilio Labra Gayo et al. "Validating RDF data". In: *Synthesis Lectures on Semantic Web: Theory and Technology* 7.1 (2017), pp. 1–328.

[12] Mauro Guerrini and Tiziana Possemato. "Linked data: a new alphabet for the semantic web". English. In: *JLIS.it* 4.1 (2013). Copyright - Copyright University of Florence, Department of Studies on the Antiquities, Middle Age, the Renaissance and Linguistics 2013; Last updated - 2018-09-24, pp. 67–70. URL: `https://www-proquest-com.ezproxy.uio.no/scholarly-journals/linked-data-new-alphabet-semantic-web/docview/1270767702/se-2?accountid=14699`.

[13] Pascal Hitzler, Markus Krtzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. 1st. Chapman and Hall/CRC, 2009. ISBN: 9781420090505.

[14] Ian Horrocks et al. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. https://www.w3.org/Submission/2004/SUBM-SWRL-20040521/. W3C, May 2004.

[15] C. Arnaud Le Hors. *RDF Data Shapes Working Group Charter*. Tech. rep. https://www.w3.org/TR shacl-20170720/. W3C, July 2017.

[16] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 359–411. ISSN: 0360-0300. DOI: 10.1145/72551.72554. URL: https://doi.org/10.1145/72551.72554.

[17] J. Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (Jan. 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf. URL: https://doi.org/10.1093/comjnl/32.2.98.

[18] David Huynh, David R Karger, Dennis Quan, et al. "Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF." In: *Semantic Web Workshop*. Vol. 52. 2002.

[19] Bart Jacobs. *Categorical logic and type theory*. eng. Amsterdam, 1999.

[20] Leif Harald Karlsen and Martin G. kjæveland. *Concepts and Abstract Model for Reasonable Ontology Templates (mOTTR)*. Oct. 2019. URL: https://spec.ottr.xyz/stOTTR/0.1/ (visited on 05/14/2021).

[21] Leif Harald Karlsen and Martin G. kjæveland. *Concepts and Abstract Model for Reasonable Ontology Templates (mOTTR)*. Mar. 2019. URL: https://spec.ottr.xyz/mOTTR/0.1/ (visited on 05/14/2021).

[22] Martin G. Kjæveland. *Batch Instantiation of OTTR templates (bOTTR)*. URL: https://spec.ottr.xyz/bOTTR/0.1/ (visited on 09/21/2021).

[23] Martin G. Kjæveland. *Tabular OTTR template instances (tabOTTR)*. URL: https://spec.ottr.xyz/tabOTTR/0.3/ (visited on 09/21/2021).

[24] Martin G. kjæveland. *Web Reasonable Ontology Templates (wOTTR)*. Dec. 2020. URL: https://spec.ottr.xyz/wOTTR/0.4/ (visited on 03/02/2022).

[25] Martin G. Kjæveland and Leif Harald Karlsen. *Adapting Reasonable Ontology Templates to RDF (rOTTR)*. URL: https://spec.ottr.xyz/rOTTR/0.2/ (visited on 12/09/2021).

[26] Holger Knublauch and Dimitris Kontokostas. *Shapes Constraint Language (SHACL)*. W3C Recommendation. https://www.w3.org/TR/2017/REC-shacl-20170720/. W3C, July 2017.

[27] Holger Knublauch and Pano Maria. *SHACL JavaScript Extensions*. https://www.w3.org/TR/2017 shacl-js-20170608/. W3C, June 2017.

[28] Lukasz Lachowski. "On the Complexity of the Standard Translation of Lambda Calculus into Combinatory Logic". eng. In: *Reports on mathematical logic* 53.53 (2018), p. 23. ISSN: 0137-2904.

[29] Daniel P Lupp, Melinda Hodkiewicz, and Martin G Skjæveland. "Template libraries for industrial asset maintenance: A methodology for scalable and maintainable ontologies". eng ; nor. In: *CEUR Workshop Proceedings*. Vol. 2757. Technical University of Aachen, 2020, pp. 49–64.

[30] Matthieu Mosser et al. "Querying APIs with SPARQL". eng. In: *Information systems (Oxford)* 105 (2022), p. 101650. ISSN: 0306-4379.

[31] Rob Nederpelt and Herman Geuvers. "Simply typed lambda calculus". In: *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014, pp. 33–68. DOI: 10.1017/CBO9781139567725.005.

[32] Terence (Terence John) Parr. *The definitive ANTLR 4 reference*. eng. Dallas, Texas, 2012.

[33] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and complexity of SPARQL". eng. In: *ACM transactions on database systems* 34.3 (2009), pp. 1–45. ISSN: 0362-5915.

[34] Axel Polleres, Paula Gearon, and Alexandre Passant. *SPARQL 1.1 Update*. W3C Recommendation. https://www.w3.org/TR/2013/REC-sparql11-update-20130321/. W3C, Mar. 2013.

[35] *Project Information*. URL: https://projects.apache.org/project.html?jena (visited on 01/24/2022).

[36] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. W3C, Jan. 2008.

[37] *Reasonable Ontology Templates (OTTR)*. URL: https://ottr.xyz/ (visited on 09/21/2021).

[38] *Reasonable Ontology Templates (OTTR)*. URL: https://ottr.xyz/#Benefits (visited on 09/21/2021).

[39] Juan L Reutter, Adrian Soto, and Domagoj Vrgoc. "Recursion in SPARQL". eng. In: *Semantic Web* 12.5 (2021), pp. 711–740. ISSN: 1570-0844.

[40] Raul Rojas. "A Tutorial Introduction to the Lambda Calculus". eng. In: (2015).

[41] M. Schönfinkel. "Über die Bausteine der mathematischen Logik". In: *Mathematische Annalen* 92 (1924), pp. 305–316. DOI: 10.1007/BF01448013. URL: https://link.springer.com/content/pdf/10.1007/BF01448013.pdf (visited on 04/27/2022).

[42] Andy Seaborne and Steven Harris. *SPARQL 1.1 Query Language*. W3C Recommendation. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/. W3C, Mar. 2013.

[43] Joshua Shinavier. "Functional programs as linked data". eng. In: *CEUR Workshop Proceedings*. Vol. 248. 2007.

[44] Martin G. Skjæveland et al. "Practical Ontology Pattern Instantiation, Discovery, and Maintenance with Reasonable Ontology Templates". In: *The Semantic Web – ISWC 2018*. Ed. by Denny Vrandečić et al. Cham: Springer International Publishing, 2018, pp. 477–494. ISBN: 978-3-030-00671-6.

[45] Martin G Skjæveland et al. "OTTR: Formal Templates for Pattern-Based Ontology Engineering". In: *Advances in Pattern-Based Ontology Engineering* 51 (2021), p. 349.

[46] Steve Bratt. *Semantic web, and other technologies to watch.* URL: https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24) (visited on 05/05/2022).

Appendix A

Formal Descriptions of Frog's Syntaxes

A.1 RDF syntax

The following two sections contain the OWL ontology for Frog's RDF syntax and the SHACL shapes defining the grammar. Additionally, we have described the OWL ontology in tables, one for the classes and the properties. The definition column in the tables represent `skos:definition`.

A.1.1 OWL vocabulary

Table A.1: Frog's RDF syntax classes vocabulary

Class IRI	Definition
frog:Function	An **function** specifies that permissible input for function calls and a function body containing function calls that can be substituted and executed
frog:functionCall	The first element of a list that should be interpreted as a function call
frog:typeArgs	The first element of a list that should be interpreted as list of generic arguments
frog:functionTerm	The first element of a list that should be interpreted as list of generic arguments
frog:lambda	The first element of a list that should be interpreted as a list containing a list of parameters as the second element, and a function call as the third element

Table A.2: Frog's RDF syntax properties vocabulary

Class IRI	Domain	Range	Definition
frog:type	frog:Function		Associates a function with the type of function
frog:parameterTypes		List of (List of (rdfs:Resource))*	Associates the type with a list containing the parameter types
frog:returnType		List of (rdfs:Resource)*	Associates the type with the return type of the function
frog:returnType		List of (rdfs:Resource)*	Associates the type with the return type of the function
frog:def	frog:Function	rdf:List	Associates the function with the list containing the lambda, with the parameter variables and function body
frog:typeVarsOf	frog:Function	rdf:List	Associates the function with the list containing the generic parameters
frog:subtypeOf		List of (rdfs:Resource)*	Associates a generic parameter with the type it is a subtype of
frog:var		(rdfs:Resource)*	Associates a generic parameter with the generic variable

```
@prefix cc: <http://creativecommons.org/ns#> .
```

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ottr: <http://ns.ottr.xyz/0.4/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix vann: <http://purl.org/vocab/vann/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix frog: <http://ns.frog.ottr.xyz/0.1#> .

frog:Function skos:definition """An *function* specifies that
                              permissible input for function calls and
                              a function body containing function calls
                              that can be substituted and executed""" .

frog:functionCall skos:definition """The first element of a list that
                                 should be interpreted as a
                                 function call""" .
frog:typeArgs skos:definition """The first element of a list that should
                             be interpreted as list of
                             generic arguments""" .
frog:lambda skos:definition """The first element of a list that should be
                             interpreted as a list containing a list of
                             parameters as the second element, and a
                             function call as the third element""".
frog:functionTerm skos:definition """The first element of a list that should
                               be interpreted as a function term""".

frog:type a owl:AnnotationProperty ;
       rdfs:domain frog:Function ;
       skos:definition """Associates a function
                       with the type of function""" .

frog:parameterTypes a owl:AnnotationProperty ;
       rdfs:range rdf:List ;
       skos:definition """Associates the type with
                        a list containing the parameter types""" .

frog:returnType a owl:AnnotationProperty ;
       skos:definition """Associates the type with the
                         return type of the function""" .

frog:def a owl:AnnotationProperty ;
       rdfs:domain frog:Function ;
       rdfs:range rdf:List ;
       skos:definition """Associates the function with the list containing
                         the lambda, with the parameter variables
```

```
                        and function body""" .

frog:typeVars a owl:AnnotationProperty ;
       rdfs:domain frog:Function ;
       rdfs:range rdf:List ;
       skos:definition "Associates the function with the list
                       containing the generic parameters" .

frog:subtypeOf a owl:AnnotationProperty ;
       skos:definition """Associates a generic parameter with
                        the type it is a subtype of""" .

frog:var a owl:AnnotationProperty ;
        skos:definition "Associates a generic parameter with the generic variable" .
```

A.1.2 SHACL shapes

```
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

@prefix skos:  <http://www.w3.org/2004/02/skos/core#> .
@prefix vann:  <http://purl.org/vocab/vann/> .
@prefix dc:  <http://purl.org/dc/elements/1.1/> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix cc: <http://creativecommons.org/ns#> .
@prefix sh:  <http://www.w3.org/ns/shacl#> .
@prefix shsh: <http://www.w3.org/ns/shacl-shacl#> .
@prefix ottr:  <http://ns.ottr.xyz/0.4/> .
@prefix o-wottr: <http://spec.ottr.xyz/wOTTR/0.4/tpl/> .
@prefix frog: <http://ns.frog.ottr.xyz/0.1#> .

<> owl:imports
    <http://www.w3.org/ns/shacl-shacl> ,
    <http://spec.ottr.xyz/rOTTR/0.2/types.shacl.ttl> .

frog:FunctionShape a sh:NodeShape;
    sh:targetClass frog:Function;
    sh:targetSubjectsOf frog:type, frog:def, frog:typeVars;
    sh:class frog:Function;
    sh:nodeKind sh:IRI;
    sh:property
          [sh:path frog:type ;
           sh:minCount 1;
           sh:maxCount 1;
           sh:node frog:TypeShape
```

```
    ],
          [sh:path frog:def ;
           sh:minCount 1;
           sh:maxCount 1;
           sh:node frog:DefinitionShape
    ],
          [sh:path frog:typeVars ;
           sh:maxCount 1;
           sh:node frog:TypeVarsShape
    ].

frog:TypeVarsShape a sh:NodeShape;
    sh:targetObjectsOf frog:typeVars;
    sh:node shsh:ListShape;
    sh:property [
        sh:path ([sh:zeroOrMorePath rdf:rest]
                rdf:first);
        sh:node frog:GenericParameterShape;
    ].

frog:GenericParameterShape a sh:NodeShape;
    sh:targetSubjectsOf frog:var, frog:subtypeOf;
    sh:property[
        sh:path frog:var;
        sh:minCount 1;
        sh:maxCount 1;
        #The shape for a generic variable
        #This shape node needs to be a blank node
        sh:node frog:GenericVariableShape;
        sh:name "generic variable";
        sh:message "a generic parameter must
                    be a blank node";
    ],
    [
        sh:path frog:subtypeOf;
        sh:minCount 1;
        sh:maxCount 1;
        #The shape for valid Frog types
        sh:node frog:TypeListShape;
    ];
    sh:name "generic parameter";
    sh:message "a generic parameter needs a
                blank node and type.".

frog:TypeShape a sh:NodeShape ;
    sh:targetSubjectsOf frog:returnType, frog:parameterTypes;
    sh:targetObjectsOf frog:type;
    sh:nodeKind sh:BlankNode;
```

```
    sh:property
          [sh:path frog:returnType;
           sh:minCount 1;
           sh:maxCount 1;
           sh:node frog:TypeListShape;
           sh:name "Return type";
           sh:message "Every type needs one return type"
    ],
          [sh:path frog:parameterTypes;
           sh:minCount 1;
           sh:maxCount 1;
           sh:node shsh:ListShape;
           sh:property [
                      sh:path ([ sh:zeroOrMorePath rdf:rest] rdf:first);
                      sh:node frog:TypeListShape
                 ];
           sh:name "Parameter types";
           sh:message "Every type needs exactly (possibly empty)
                      list of parameter types"
    ].

frog:DefinitionShape a sh:NodeShape;
     sh:targetObjectsOf frog:def ;
     sh:node shsh:ListShape ;
     sh:property [
          sh:path rdf:first ;
          sh:hasValue frog:lambda ;
          sh:minCount 1
      ],
     [
          sh:path (rdf:rest rdf:first);
          sh:node frog:ParameterListShape ;
          sh:minCount 1 ;
      ],
     [
          sh:path (rdf:rest rdf:rest rdf:first);
          sh:node frog:FunctionCallShape ;
          sh:minCount 1;
      ],
     [
          sh:path (rdf:rest rdf:rest rdf:rest);
          sh:hasValue rdf:nil ;
          sh:minCount 1;
      ].

frog:ParameterListShape a sh:NodeShape ;
     sh:node shsh:ListShape ;
     sh:property
```

```
            [sh:path ([sh:zeroOrMorePath rdf:rest ] rdf:first );
             sh:nodeKind sh:BlankNode;
      ].


frog:FunctionCallShape a sh:NodeShape ;
    sh:node shsh:ListShape ;
    sh:property [
        sh:path rdf:first ;
        sh:hasValue frog:functionCall ;
        sh:minCount 1
    ],
    [
        sh:path (rdf:rest rdf:first) ;
        sh:nodeKind sh:BlankNodeOrIRI ;
        sh:minCount 1
    ] ;
    sh:or (
        [
        sh:property [
            sh:path (rdf:rest rdf:rest rdf:first );
            sh:minCount 1 ;
            sh:node frog:GenericArgumentsShape
        ],
        [
            sh:path (rdf:rest rdf:rest
                       [sh:oneOrMorePath rdf:rest] rdf:first ) ;
            sh:name "Arguments"
        ]
    ]
    [
        sh:property [
                   sh:path (rdf:rest [sh:oneOrMorePath rdf:rest]
                             rdf:first ) ;
                   sh:name "Arguments"
               ] ;
        sh:not [
               sh:property [
                      sh:path (rdf:rest rdf:rest rdf:first );
                      sh:node shsh:ListShape ;
                      sh:property [
                                    sh:path rdf:first;
                                    sh:hasValue frog:typeArgs
                                ]
                   ]
               ]
        ]
     ) .
```

```
frog:GenericArgumentsShape a sh:NodeShape;
    sh:node shsh:ListShape;
    sh:property
          [
              sh:path rdf:first ;
              sh:hasValue frog:typeArgs;
              sh:minCount 1;
           ],
              [
              sh:path ([ sh:oneOrMorePath rdf:rest] rdf:first);
              sh:node frog:TypeListShape;
           ].

frog:TypeListShape a sh:NodeShape ;
    sh:targetObjectsOf frog:returnType, frog:subtypeOf;
    sh:xone
          (
        [ sh:node ottr:ListTypeShape ;
          sh:name "List type" ;
          sh:message """Unrecognised list type. A list a list of types,
           where the last item in the list must be a basic type, the second
           last can be a 'least upper bound'
            type, and the types preceeding it can be list types."""
        ]
        [ sh:node ottr:LUBTypeShape, shsh:ListShape ;
          sh:name "List type" ;
          sh:message """Unrecognised lub type. A LUB a list of types,
               a list of the LUB iri and basic/generic type."""
        ]
        [ sh:node ottr:FunctionTypeShape ;
          sh:name "Function type" ;
          sh:message """Unrecognised lub type. A LUB a list of types,
           a list of the LUB iri and basic/generic type."""
        ]
        [ sh:node ottr:BasicTypeShape ;
          sh:name "Basic parameter type" ;
          sh:message "Unrecognised basic type. A type is specified
          either as an RDF list of types or a single basic type."
        ]
        [
            sh:node frog:GenericVariableShape ;
            sh:name "Generic type" ;
            sh:message "Unrecognised generic type. A type is specified
            either as an RDF list of types or a single basic type."
        ]
     ).
```

```
frog:GenericVariableShape a sh:NodeShape;
    sh:nodeKind sh:BlankNode ;
    sh:name "Generic variable";
    sh:property
         [ sh:path ( [ sh:zeroOrMorePath rdf:rest ] rdf:first ) ;
           sh:minCount 0;
           sh:maxCount 0;
    ] .

ottr:ListTypeShape a sh:NodeShape ;
    sh:node shsh:ListShape ;
    sh:or ( # Last value is a function, generic or base type
        [sh:property [
              sh:path [sh:zeroOrMorePath rdf:rest ];
              sh:or (
                  [sh:hasValue rdf:nil]
                  [
                      sh:property [
                          sh:path rdf:first;
                           sh:or ([sh:hasValue rdf:List]
                                  [sh:hasValue ottr:NEList])
                      ];
                      sh:not[a sh:PropertyShape;
                              sh:path rdf:rest;
                              sh:hasValue rdf:nil]
                  ]
                  [
                      sh:property
                          [sh:path rdf:first;
                          sh:node ottr:BasicGenericAndFunctionShape],
                          [sh:path rdf:rest;
                          sh:hasValue rdf:nil ]
                  ]
              )
           ]
        ]
        [ #last value is a lub type
            sh:property [
              sh:path [sh:zeroOrMorePath rdf:rest ];
              sh:or (
                  [sh:hasValue rdf:nil]
                  [
                      sh:property [sh:path rdf:first;
                                   sh:or ([sh:hasValue rdf:List]
                                  [sh:hasValue ottr:NEList])
                                  ];
                      sh:not[a sh:PropertyShape;
                              sh:path rdf:rest;
```

```
                            sh:hasValue rdf:nil
                            ]
                ]
                [
                    sh:property
                        [sh:path rdf:first; sh:minCount 1;
                        sh:hasValue ottr:LUB],
                        [sh:path (rdf:rest rdf:first); sh:minCount 1;
                         sh:node ottr:BasicGenericAndFunctionShape],
                        [sh:path (rdf:rest rdf:rest); sh:minCount 1;
                         sh:hasValue rdf:nil]
                ]
                [
                    sh:property
                        [sh:path rdf:first; sh:minCount 1;
                         sh:node ottr:BasicGenericAndFunctionShape],
                        [sh:path rdf:rest; sh:minCount 1;
                         sh:hasValue rdf:nil]
                ]
            )
        ]
    ]
 ).

ottr:FunctionTypeShape a sh:NodeShape ;
    sh:node shsh:ListShape ;
    sh:property [
        sh:path rdf:first;
        sh:hasValue frog:Function;
        sh:minCount 1
     ],
    [
        sh:path ([sh:oneOrMorePath rdf:rest]
                rdf:first);
        sh:minCount 1;
        #The shape for valid Frog types
        sh:node frog:TypeListShape
     ].


ottr:LUBTypeShape a sh:NodeShape ;
    sh:property [
        sh:path rdf:first ;
        sh:minCount 1;
        sh:hasValue ottr:LUB
    ],
    [
        sh:path (rdf:rest rdf:first);
```

```
        sh:minCount 1;
        sh:node frog:TypeListShape
    ],
    [
        sh:path (rdf:rest rdf:rest);
        sh:minCount 1;
        sh:hasValue rdf:nil
    ].

ottr:BasicAndGenericShape a sh:NodeShape ;
    sh:xone (
        ottr:BasicTypeShape
        frog:GenericVariableShape
    ).

ottr:BasicGenericAndFunctionShape a sh:NodeShape ;
    sh:xone (
        ottr:BasicAndGenericShape
        ottr:FunctionTypeShape
    ).
```

A.2 Human Readable Syntax

The following file is the ANTLR4 grammar formally describing the syntax of Frog's HRS and stOTTR:

```
1  grammar stOTTR;
2
3  import Turtle;
4
5  stOTTRDoc
6   : ( directive // Turtle prefixes and base
7     | statement )* EOF
8   ;
9
10 statement
11  : ( signature
12      | template
13      | baseTemplate
14      | instance
15    )
16  '.'
17  ;
18
19
20 /*** Comments ***/
21
22 Comment
23  : '#' ~('\r' | '\n')* -> skip
```

```
24  ;
25
26  CommentBlock
27   : '/***' .*? '***/' -> skip
28   ;
29
30
31  /*** Signature ***/
32
33  signature
34   : templateName parameterList annotationList?
35   ;
36
37  templateName
38   : iri
39   ;
40
41  parameterList
42   : '[' (parameter (',' parameter)*)? ']'
43   ;
44
45  parameter
46   : ParameterMode* type? Variable defaultValue?
47   ;
48
49  ParameterMode
50   : '?' /* optional */
51   | '!' /* non blank */
52   ;
53
54  defaultValue
55   : '=' constant
56   ;
57
58  annotationList
59   : annotation (',' annotation)*
60   ;
61
62  annotation
63   : '@@' instance
64   ;
65
66
67  /*** Templates ***/
68
69  baseTemplate
70   : signature '::' 'BASE'
71   ;
```

```
72
73 template
74  : signature '::' patternList
75  ;
76
77 patternList
78  : '{' (instance (',' instance)*)? '}'
79  ;
80
81
82 /*** Instance ***/
83
84 instance
85  : (ListExpander '|')? templateName argumentList
86  ;
87
88 ListExpander
89  : 'cross'
90  | 'zipMin'
91  | 'zipMax'
92  ;
93
94 argumentList
95  : '(' (argument (',' argument)*)? ')'
96  ;
97
98 argument
99  : ListExpand? term
100  ;
101
102 ListExpand
103  : '++'
104  ;
105
106
107 /*** Types ***/
108
109 type
110  : basicType
111  | lubType
112  | listType
113  | neListType
114  | functionType
115  | genericType
116  ;
117
118  functionType
119  : 'Function<'((type ',')* type) '>'
```

```
120  ;
121
122 genericType
123     : Variable
124     ;
125
126 listType
127  : 'List<' type '>'
128  ;
129
130 neListType
131  : 'NEList<' type '>'
132  ;
133
134 lubType
135  : 'LUB<' (basicType | genericType) '>'
136  ;
137
138 basicType
139  : prefixedName
140  ;
141
142
143 /*** Terms ***/
144
145 term
146  : Variable
147  | constant
148  | list
149  | functionCall
150  | functionTerm
151  ;
152
153 functionCall
154     : '(' functionCallDef? functionCallName genericArguments? term* ')'
155     ;
156
157 functionCallName
158     : iri
159     | Variable ;
160
161 functionCallDef : iri ;
162
163 genericArguments
164     : '<' '<' (genericArgument (',' genericArgument)*)? '>' '>'
165     ;
166
167 genericArgument
```

```
168     : Variable
169     | type
170     ;
171
172 Variable
173  : '?' BNodeLabel
174  ;
175
176 /* Turtle blank node labels without trailing '_:' */
177 fragment BNodeLabel
178  : (PN_CHARS_U) ((PN_CHARS | '.')* PN_CHARS)?
179  ;
180
181 constant
182  : iri
183  | blankNode
184  | literal
185  | none
186  ;
187
188 none
189  : 'none'
190  ;
191
192 list
193  : '(' (term (',' term)*)? ')'
194  ;
195
196 functionTerm
197  : iri genericArguments
198  ;
199
200
201 /*** Frog ***/
202 frogDoc
203     : ( directive
204     | function
205     | functionCall )* EOF ;
206
207 function
208     : functionHead '::' functionBody '.'
209     ;
210
211 functionHead
212     : definition genericParameterList* frogParameterList returnType
213     ;
214
215 definition
```

```
216     : 'def' name
217     ;
218
219 name
220     : iri
221     ;
222
223 genericParameterList
224     : '<' '<'
225         ((genericParameter ',')* genericParameter?)
226       '>' '>'
227     ;
228
229 genericParameter
230     : Variable 'subtypeOf' type
231     ;
232
233 frogParameterList
234     : '(' (frogParameter (',' frogParameter)*)? ')'
235     ;
236
237 frogParameter
238     : type Variable
239     ;
240
241 returnType
242     : '->' type
243     ;
244
245 functionBody
246     : functionCall
247     ;
```

Appendix B

Validation queries

Prefixes used in this appendix:

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xs:   <http://www.w3.org/2001/XMLSchema#>
PREFIX frog: <http://ns.frog.ottr.xyz/0.1#>
```

B.1 Function defined

```
SELECT DISTINCT ?functionName ?functionCallName
WHERE {
    #finds everything used as a function call
    #name in the function body
    ?functionName a frog:Function;
       frog:body/(frog:arg/frog:val)*/frog:of ?functionCallName.

    #removes all mathces where the function call
    #name is an IRI and the IRI is of type function
    FILTER NOT EXISTS {
            ?functionCallName a frog:Function.
            FILTER isIRI(?functionCallName)
    }
    #removes all mathces where a blank node is used as
    #function is defined as a parameter of type function
    FILTER NOT EXISTS {
            ?functionName frog:parameter
                          [frog:var ?functionCallName;
                           frog:parameterType [a frog:Function]
                          ]
            FILTER isBlank(?functionCallName)
    }
}
```

B.2 Undefined parameter variable

```
SELECT DISTINCT ?functionName ?parameterVariable
```

```
WHERE {
    #Finds every parameter variable used as in function body
    ?functionName a frog:Function.
    { #parameter used as an argument (also in a list)
        ?functionName frog:body/(frog:arg/frog:val)+/
                      (rdf:rest*/rdf:first)* ?parameterVariable
    } UNION { #parameter used as a function
        ?functionName frog:body/(frog:arg/frog:val)*/
                                frog:of ?parameterVariable
    }

    FILTER isBlank(?parameterVariable)
    FILTER NOT EXISTS{ #remove defined parameters
        ?functionName frog:parameter/frog:var ?parameterVariable
    }

    #NEXT TO NOT EXISTS REMOVES BLANK NODES THAT ARE NOT VARIABLES
    FILTER NOT EXISTS { #remove list blank nodes
        ?parameterVariable rdf:first [];
            rdf:rest [].
    }

    FILTER NOT EXISTS { #remove function calls blank nodes
        ?parameterVariable frog:of [].
    }
}
```

B.3 Undefined generic parameter variable

```
SELECT ?functionName ?genericVariable
WHERE {
    ?functionName a frog:Function;
                  frog:body []. #only finds Frog functions.

    { #Used as a generic argument
        ?functionName frog:body/(frog:arg/frog:val)*/frog:typeArg/
        (frog:type/(frog:argType+/frog:type)+)? ?genericArgument
    } UNION { #used on return type
        ?functionName frog:returnType/
                      (frog:argType+/frog:type)* ?genericArgument
    } UNION { #used in parameter
        ?functionName frog:parameter/frog:parameterType/
                      (frog:argType+/frog:type)* ?genericArgument
    }

    ?genericArgument a frog:GenericType;
                     frog:type ?genericVariable.
    FILTER NOT EXISTS{
        ?functionName frog:typeVar/frog:var ?genericVariable.
    }

```

```
}
```

B.4 Correct arity arguments

```
SELECT DISTINCT *
WHERE{
    ?functionName a frog:Function;
        frog:body/(frog:arg/frog:val)* ?functionCall.
    ?functionCall frog:of ?functionCallName.

    { #finds how many arguments the function call has
        SELECT ?functionCall (COUNT(?rec) AS ?received)
        WHERE {
            ?functionCall frog:of [].
            OPTIONAL{
                ?functionCall frog:arg ?rec.
            }
        }
        GROUP BY ?functionCall
    }
    #finds how many parameters the function has
    { #if the function is defined with a parameter variable
        {
            SELECT ?functionCallName (COUNT(?exp) AS ?expected)
            WHERE {
                    [] a frog:Function;
                        frog:parameter [
                            frog:var ?functionCallName;
                            frog:parameterType ?parType
                        ].
                    ?parType a frog:Function.
                    OPTIONAL { #finds the parameters
                               #of a parameterfunction
                        ?parType frog:argType/frog:index ?exp
                    }
            }
            GROUP BY ?functionCallName
        }
    } UNION { #if the functon is defined with a IRI
        {
            SELECT ?functionCallName (COUNT(?exp) AS ?expected)
            WHERE {
                ?functionCallName a frog:Function.

                OPTIONAL{
                    ?functionCallName frog:parameter ?exp.
                }
                FILTER isIRI(?functionCallName)
            }
            GROUP BY ?functionCallName
        }
```

```
48      }
49     FILTER(?received != ?expected)
50  }
51  ORDER BY ?functionName ?functionCallName
```

B.5 Correct arity generic arguments

```
1   SELECT *
2   WHERE {
3       ?functionName a frog:Function;
4           frog:body/(frog:arg/frog:val)* ?functionCall.
5
6       ?functionCallName a frog:Function.
7
8       #finds how many generic argument a function call has
9       {
10          {
11              SELECT ?functionCall ?functionCallName (COUNT(?rec) AS ?received)
12              WHERE {
13                  ?functionCall a frog:functionCall;
14                                frog:of ?functionCallName.
15                  OPTIONAL{
16                      ?functionCall frog:typeArg ?rec.
17                  }
18              }
19              GROUP BY ?functionCall ?functionCallName
20          }UNION{ #Function used as arguments
21              SELECT ?functionCall ?functionCallName (COUNT(?rec) AS ?received)
22              WHERE{
23                  ?thisFunctionName a frog:Function;
24                            frog:body/(frog:arg/frog:val)* ?functionCall.
25                  ?functionCall a frog:functionCall;
26                                frog:of ?outerFunctionCall;
27                                frog:arg [frog:index ?index;
28                                          frog:val/(rdf:rest*/rdf:first)* ?functionCallName
29                                        ].
30                  #CHECK THAT ?functionCallName SHOULD BE A FUNCTION
31                  { #Function name is an IRI and parameter is a function
32                      ?outerFunctionCall frog:parameter [ frog:index ?index;
33                                                          frog:parameterType/rdf:type frog:Function
34                                                        ].
35                      FILTER isIRI(?outerFunctionCall)
36                  }UNION { #Function name is an parameter variable and parameter is a function
37                      ?thisFunctionName frog:parameter [ frog:var ?outerFunctionCall;
38                                                         frog:parameterType[
39                                                                 frog:argType [a frog:Function; frog:index ?index]
40                                                             ]
41
42                                                        ].
43                      FILTER isBlank(?outerFunctionCall)
44                  } UNION{ #Function name is an IRI and parameter is a list of functions
45                      SELECT ?outerFunctionCall ?index
46                      WHERE {
47                          VALUES ?listIRI {rdf:List ottr:NEList}
48                          ?outerFunctionCall frog:parameter [ frog:index ?index;
49                                                              frog:parameterType [a ?listIRI; frog:argType+ ?innerMost];
50                                                              ].
51                          ?innerMost a frog:Function.
52                          FILTER NOT EXISTS{
53                              ?innerMost a ?listIRI
54                          }
55
56                          FILTER NOT EXISTS{ #Remove the function inside a function
57                              ?innerMost ^frog:argType+ [a frog:Function].
58                          }
59                      }
60                  } UNION { #Function name is an parameter variabel and parameter is a list of functions
61                      SELECT ?thisFunctionName ?index ?outerFunctionCall
62                      WHERE {
63                          VALUES ?listIRI {rdf:List ottr:NEList}
64                          ?thisFunctionName frog:parameter [ frog:var ?outerFunctionCall;
65                                                             frog:parameterType [a frog:Function;
66                                                                                 frog:argType [
67                                                                                     a ?listIRI;
68                                                                                     frog:argType+ ?innerMost;
69                                                                                     frog:index ?index
70                                                                                     ];
71
72                                                                                 ];
73                                                            ].
74                          ?innerMost a frog:Function.
75                          FILTER NOT EXISTS{
76                              ?innerMost a ?listIRI
77                          }
78
79                          FILTER NOT EXISTS{ #Remove the function inside a function, execept the first
80                              ?innerMost ^frog:argType+ ?invers.
81                              ?invers a frog:Function;
82                              FILTER NOT EXISTS{
83                                  [] frog:parameterType ?invers
84                              }
85                          }
86                      }
```

```
87                 }
88
89                 FILTER isIRI(?functionCallName)
90             }
91             GROUP BY ?functionCall ?functionCallName ?functionCallName
92         } UNION { #FUNCTION IRI
93             SELECT ?functionCall ?functionCallName  (COUNT(?rec) AS ?received)
94             WHERE {
95                 ?functionCall a frog:functionCall;
96                             frog:arg/frog:val/(rdf:rest*/rdf:first)* ?potentialFunction.
97
98                 ?potentialFunction a frog:Function;
99                             frog:of ?functionCallName.
100                OPTIONAL{
101                    ?potentialFunction frog:typeArg ?rec
102                }
103            }
104            GROUP BY ?potentialFunction ?functionCall ?functionCallName
105        }
106    }
107    #find hoe many generic arguments the function expects
108    { # function defined by IRI
109        SELECT ?functionCallName (COUNT(?exp) AS ?expected)
110        WHERE {
111            ?functionCallName a frog:Function.
112
113            OPTIONAL{
114                ?functionCallName frog:typeVar ?exp.
115            }
116            FILTER isIRI(?functionCallName)
117        }
118        GROUP BY ?functionCallName
119    }
120    FILTER(?received != ?expected)
121 }
122 ORDER BY ?functionName ?functionCallName
```

B.6 Unused parameter

```
1  SELECT ?functionName ?parameterVariable
2  WHERE {
3      ?functionName a frog:Function;
4                    frog:parameter [frog:var ?parameterVariable];
5                    frog:body []. #only Frog functions
6     FILTER isBlank(?parameterVariable)
7
8     #finds every parameter variable used as an argument(also lists)
9     FILTER NOT EXISTS {
10        ?functionName frog:body/(frog:arg/frog:val)+/
11                      (rdf:rest*/rdf:first)* ?parameterVariable
12    }
13    # finds every parameter varaibel used as an function
14    FILTER NOT EXISTS {
15        ?functionName frog:body/(frog:arg/frog:val)*/
16                                  frog:of ?parameterVariable
17    }
18 }
```

B.7 Unused generic parameter variable

```
1  SELECT ?functionName ?genericVariable
2  WHERE {
3      ?functionName a frog:Function;
4                    frog:typeVar/frog:var ?genericVariable;
5                    frog:body []. #only Frog functions
6     FILTER isBlank(?genericVariable)
7
8     FILTER NOT EXISTS { #In parameter
```

```
9           ?functionName frog:parameter/frog:parameterType/
10                        (frog:argType+/frog:type)* [a frog:GenericType;
11                                                    frog:var ?genericVariable].
12      }
13
14      FILTER NOT EXISTS { #In return type
15          ?functionName frog:returnType/
16                       (frog:argType+/frog:type)* [a frog:GenericType;
17                                                  frog:var ?genericVariable].
18      }
19
20      FILTER NOT EXISTS { #In generic argument
21          ?functionName frog:body/(frog:arg/frog:val)*/
22                        frog:typeArg/(frog:type/
23                        (frog:argType+/frog:type)+)? [a frog:GenericType;
24                                                     frog:type ?genericVariable]
25      }
26
27  }
```

Appendix C

Timing of OTTR execution with and without Frog

This appendix contains the times used to create Figure 6.11 in Section 6.2.3, which compares the time used to expand instances with and without Frog. Note that the times are in seconds.

Without Frog

	100	1000	10000	100000
1	2	4	14	164
2	2	4	17	168
3	2	4	17	168
4	3	4	17	178
5	3	5	17	174
6	3	5	17	173
7	4	5	17	171
8	2	5	26	176
9	3	5	22	181
10	3	5	15	178
Avg	2,7	4,6	17,9	173,1

With Frog

	100	1000	10000	100000
1	5	7	34	479
2	4	8	37	498
3	5	8	36	522
4	5	9	35	507
5	5	10	35	512
6	7	9	35	518
7	6	9	35	506
8	5	9	35	517
9	6	9	49	510
10	5	10	33	507
Avg	5.3	8.8	36.4	507.6

Average of with and without Frog

	100	1000	10000	100000
Without Frog	2,7	4,6	17,9	173,1
With Frog	5.3	8.8	36.4	507.6